LABORATORIO DE COMUNICACIONES DIGITALES EN PYTHON

Editorial Universidad de Sevilla

F. Javier Payán Somet
Juan José Murillo Fuentes
José Carlos Aradillas

LABORATORIO DE COMUNICACIONES DIGITALES EN PYTHON

Editorial Universidad de Sevilla

Escuela Técnica Superior de
INGENIERÍA DE SEVILLA

SEVILLA 2022

Colección: Monografías de la Escuela Técnica Superior de Ingeniería
de la Universidad de Sevilla

Núm.: 23

Motivo de cubierta: Camaleón. Imagen de Jean Photosstock en Pixabay

C/ Porvenir, 27 - 41013 Sevilla.
Tlfs.: 954 487 447; 954 487 451; Fax: 954 487 443
Correo electrónico: eus4@us.es
Web: <https://www.editorial.us.es>

Impreso en papel ecológico
Impreso en España-Printed in Spain

ISBN 978-84-472-2273-5
Depósito Legal: SE 1310-2022

Diseño de interior (LaTeX): F. Javier Payán Somet 2014.
Maquetación de interior (LaTeX): F. Javier Payán
Diseño de cubierta: Santi García Hernández
Realización de cubierta: referencias.maquetacion@gmail.com

Impresión: Masquelibros

Prefacio

La descripción adecuada de una teoría debe ser lo más simple posible. Pero no demasiado simple.

Albert Einstein

Este texto constituye el Manual de Prácticas de Laboratorio de la asignatura Comunicaciones Digitales del tercer curso del Grado de Ingeniería de las Tecnologías de Telecomunicación impartido en la Escuela Técnica Superior de Ingeniería de la Universidad de Sevilla durante el curso 2020/21.

En las prácticas presentadas se simulan, mediante un software de cálculo numérico, temas afines al campo de las Comunicaciones Digitales, de forma que el alumno profundice en los conceptos de la asignatura desde un punto de vista más práctico. El código suministrado está pensado para ejecutarse en cualquier intérprete del lenguaje de programación Python con una versión posterior a 3.X. De forma alternativa existe el manual donde el código se suministra en MATLAB, que es el lenguaje que se ha utilizado durante los años anteriores en la asignatura.

La metodología desarrollada en cada uno de los capítulos es común: se plantea el objetivo a alcanzar, se realiza una exposición más o menos extensa en la que se presentan códigos y numerosos ejemplos a la vez que se proponen una serie de ejercicios directamente relacionados con lo anterior que el alumno debería haber completado para una asimilación de lo estudiado. En la sesión de laboratorio se trabaja de forma dirigida en la solución de un ejercicio planteado. Es decir, se supone que la exposición ha sido estudiada por el alumno y la sesión de laboratorio estará dedicada a la resolución *por parte del alumno*, con la ayuda del profesor, de un ejercicio propuesto.

El Capítulo 1 repasa los aspectos más sobresalientes del lenguaje de programación Python. Se presta una especial atención a la relación existente entre las señales continuas en el tiempo y las secuencias discretas que la describen.

En el Capítulo 2 se profundiza en el conocimiento de variables aleatorias. En Python existen diversos comandos que permiten generar valores de una variable aleatoria con una determinada función densidad de probabilidad y, haciendo uso de los mismos, se generan los valores de las variables aleatorias que se desea estudiar. Se destaca la definición de las funciones que aproximan la función masa de probabilidad o función densidad de probabilidad de variables aleatorias discretas y continuas, respectivamente. Se hace especial hincapié en que el alumno distinga claramente entre los valores que una determinada variable toma en cierto experimento y la función densidad de probabilidad de la misma.

El Capítulo 3 está dedicado al estudio del test de hipótesis, por la importancia clave que tiene en el conocimiento del detector óptimo. Como ocurre en los demás casos, se plantea que el desarrollo teórico de la asignatura y el de las prácticas discurran de forma paralela, lo que en este caso es especialmente importante por la dificultad teórica que presenta una comprensión adecuada del concepto de detección y probabilidad de error.

Siguiendo el esquema que se desarrolla en la asignatura, los Capítulos 4 y 5 están dedicados al estudio de la modulación. Dada una secuencia de dígitos binarios, existen numerosas formas de realizar su transmisión digital a través de un canal. Las diversas soluciones planteadas derivan de los requerimientos de todo tipo que se le plantea al diseñador. Se propone un esquema de modulación lineal y se definen diversas funciones que representan los tres tipos de modulaciones lineales más frecuentes.

El Capítulo 6 aborda la simulación práctica de un canal AWGN, la construcción del receptor vectorial mediante correladores y filtros adaptados y, colateralmente, la constelación de las señales transmitidas. Experimentalmente se obtiene el vector de observación y se muestra el *scattering plot*, analizando con detenimiento la modificación del mismo en función de las diversas modulaciones y la potencia del ruido en el canal.

En el Capítulo 7 se presenta el estudio de un detector óptimo en un canal AWGN para señales linealmente moduladas. Se analizan diversos esquemas de modulación y se calculan y comparan las probabilidades de error tanto de símbolos como de bits teóricas y prácticas.

Por último, en el Apéndice se incluye una breve descripción de como instalar Anaconda, bien porque se prefiera la versión de Spyder que incluye, aunque se recomienda el uso del Spyder-IDE (ver https://www.spyder-ide.org/), o bien porque se quiera ejecutar de forma local los notebooks que como material adicional se aportan. Estos notebooks se pueden encontrar en https://github.com/gapsc-us/labcomdig e incluyen los códigos descritos en cada capítulo. De allí también se puede descargar la librería labcomdig.py, necesaria para desarrollar los ejercicios así como un fichero environment.yml con las versiones de los packages de python utilizados.

Los autores.

Índice

Práctica

1

Introducción a Python

The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point.

CLAUDE SHANNON, 1948

El objetivo de este primer capítulo es presentar un breve resumen del lenguaje de programación Python, mostrando sus principales características y peculiaridades y proponer un entorno en el que se puedan escribir y ejecutar los ejemplos y ejercicios correspondientes a un laboratorio de comunicaciones digitales. Como un primer ejemplo de escritura de un programa en Python se planteará un método para representar una señal continua mediante una señal discreta, de forma que la energía de ambas representaciones coincida. Esta equivalencia se mantendrá constante durante los diversos ejercicios prácticos que se irán proponiendo como parte del diseño de un sistema de comunicación digital.

1.1 Introducción

Python es un lenguaje de programación *interpretado* de propósito general creado a finales de los 80, que soporta múltiples paradigmas de programación tales como la orientada a objetos, la imperativa y la funcional [Lut13]. Entre sus características principales hay que destacar su carácter de código abierto y el ser un lenguaje multiplataforma, lo que permite ejecutar un programa Python en cualquier máquina, con cualquier sistema operativo, utilizando el mismo conjunto de líneas de código.

Python lleva varios años siendo el lenguaje de programación más utilizado en el ámbito de la ingeniería y electrónica, entre otros sectores, como se recoge en [Int] (ver https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020) y se muestra en la Tabla 1.1.

Tabla 1.1: Ranking de lenguajes de programación más utilizados en 2020. [Int].

Posición	Lenguaje	Puntuación
1	Python	100,0
2	Java	95,3
3	C	94,6
4	C++	87,0
5	JavaScript	79,5
6	R	78,6
7	Arduino	73,2
8	Go	73,1
9	Swift	70,5
10	MatLab	68,9

Si se consulta dicha lista para años anteriores, podrá observarse que Python aparece normalmente en primera posición. Este hecho, junto con el carácter de código abierto, justifica la elección de Python como lenguaje para la edición de un manual de laboratorio de Comunicaciones Digitales. Sin embargo, aunque este texto está orientado específicamente a todo aquel que desee simular sistemas de comunicación digital, las herramientas básicas que se van a desarrollar permitirá un uso en otros ámbitos académicos y profesionales.

Una de las características más atractivas de Python, aparte de la simplicidad de su sintaxis, consiste en la existencia de un ecosistema de herramientas específicas desarrolladas en una gran variedad de campos científicos [Van16a], y en ofrecer una gran cantidad de aplicaciones, paquetes y librerías en código abierto (disponibles en plataformas de distribución de software como GitHub, por ejemplo) para el desarrollo de programas en un dominio enormemente amplio. En este capítulo se propondrá, entre otras cosas, un método para instalar y utilizar algunas de estas aplicaciones, así como la instalación y utilización de librerías y paquetes adicionales, escritos con propósitos específicos, que amplían el uso básico de Python.

Actualmente existen dos versiones del lenguaje Python, la 2.X y la 3.X. En este manual se utilizará la sintaxis de Python 3, no compatible con las versiones 2.X de Python. Aunque Python 3.0 fue liberado en 2008, su adopción por la comunidad científica ha sido lenta, a causa de la gran cantidad de paquetes esenciales escritos en las versiones 2.X de Python. Sin embargo, el esfuerzo de múltiples desarrolladores y la estabilidad alcanzada por las diferentes versiones 3.X de Python hacen aconsejable centrarse exclusivamente en esta versión del lenguaje.

Es importante distinguir entre el lenguaje propiamente dicho y el programa que permite la ejecución de un código escrito en ese lenguaje. Como se ha dicho, Python es

un lenguaje interpretado y a cada versión del lenguaje le corresponde un intérprete del mismo. Este intérprete se encuentra instalado por defecto en la mayoría de los sistemas operativos y la versión del intérprete y el lenguaje en el que está escrito un determinado programa deben coincidir. Como se ha dicho, no hay compatibilidad completa entre las versiones la 2.X y la 3.X del lenguaje Python y, por tanto, la versión del intérprete que se utiliza en la ejecución de los programas escritos en las diferentes versiones deberán corresponderse con el intérprete que se invoque en el sistema operativo.

Se puede observar que en ocasiones hay más de un intérprete de Python instalado en un mismo ordenador. Así, por ejemplo, en los ordenadores con sistema operativo macOS Big Sur, hay instaladas dos versiones del intérprete, en el momento de escribir estas líneas, la 2.7 y la 3.9.4. Para invocarlos se puede teclear en una ventana del Terminal `python` ó `python3`, respectivamente. Dependiendo de la versión del sistema operativo existente en cada ordenador, Unix, Linux, Windows, etc, se accede al intérprete o intérpretes instalado de manera análoga a como se hace en macOS Big Sur.

En este texto, dentro de las versiones de Python 3, se ha utilizado en la escritura de los diferentes programas Python 3.7 y se debe garantizar que todos los módulos y librerías que se precisan para la ejecución de los programas escritos aquí sean compatibles con esta versión.

En la librería `labcomdig.py`, que se puede encontrar en https://github.com/gapsc-us/labcomdig/, se han incluido las funciones necesarias para llevar a cabo los ejercicios propuestos. Muchas de estas funciones se describirán en este texto.

1.2 Entorno de desarrollo para Python

A la hora de realizar las prácticas es necesario disponer de un entorno donde poder programar y observar los resultados. En Python esto se puede hacer bien a través de un visor o navegador web o bien utilizando un entorno integrado de desarrollo (IDE). En el primer caso `Jupyter` es un entorno ampliamente utilizado. En el segundo los IDE `Spyder` y `PyCharm` son muy conocidos.

El presente laboratorio se ha programado utilizando Spyder como entorno de desarrollo, y es la solución recomendada. Para instalar y usar Spyder existe[1] una versión que se instala fácilmente, Spyder-IDE (ver https://www.spyder-ide.org/), que es *la recomendada en este texto.* Esta versión incluye por defecto los paquetes necesarios para abordar los ejercicios propuestos (matplotlib, numpy y scipy) y ejecutar los códigos incluidos. Todos estos códigos se pueden copiar y salvar como scripts de Python y ejecutar en Spyder. Las soluciones a los ejercicios propuestos también se puede llevar a cabo en este IDE.

En la pestaña `Tools` de Spyder encontrará la opción `PYTHONPATH manager`. Puede descargar el fichero labcomdig.py de https://github.com/gapsc-us/labcomdig/ y sal-

[1] Esta versión IDE Spyder se ha sido desarrollada mientras se preparaba este texto.

varlo en cualquier carpeta. Incluya después el *path* de esta carpeta en la lista que se abre al hacer clic en esta opción PYTHONPATH manager. A partir de entonces podrá usar la librería en cualquier script, independientemente del directorio de trabajo.

1.2.1 Entorno de desarrollo Spyder

El IDE Spyder[2] consta de varias partes bien diferenciadas, de forma análoga en su apariencia a lo mostrado en la ejecución de otras herramientas similares, como MATLAB. Por un lado, incluye un editor de texto optimizado para la escritura de scripts de Python y por otro lado una ventana en la que se pueden ejecutar comandos del intérprete interactivo mejorado de Python, IPhyton. A las dos partes anteriores cabe añadirle una ventana desde la que gestionar múltiples elementos. Los scripts escritos en Python se pueden ejecutar en Spyder haciendo click en el botón play de la barra superior de herramientas, cuando están abiertos en la ventana del editor.

Al arrancar Spyder, en la ventana de consola puede aparecer algo como lo siguiente:

```
Python 3.7.10 | packaged by conda-forge | (default, Oct 5 2021, 16:59:49)
Type "copyright", "credits" or "license" for more information.

IPython 7.12.0 -- An enhanced Interactive Python.
In [1]:
```

La línea `In [1]:` señala que se pueden escribir y ejecutar líneas de código escritas en el mismo. Así, se escribe y ejecuta la típica primera sentencia en Python:

```
[1]: print("Hola mundo")
Hola mundo

[2]:
```

En lugar de escribir y ejecutar una línea de código en Python cada vez, es posible utilizar un editor de texto y escribir un conjunto de líneas de código en un fichero (texto plano), o *script*, salvarlo con la extensión `.py` e invocar el intérprete de Python para que ejecute las órdenes escritas en dicho fichero, por ejemplo, en la forma:

```
[2]: run 'nombrescript'
```

Pero la forma natural de ejecutar scripts de Python en Spyder es abrir el fichero en el editor que trae incorporado y darle a la opción de ejecutar.

Nótese que al arrancar el IDE, en la consola aparecía una cabecera que terminaba con la línea `IPython 7.12.0 -- An enhanced Interactive Python.` Este

[2] Scientific PYthon Development EnviRonment.

IPython[3] es una forma mejorada de ejecución de Python. Con IPython es fácil generar figuras al ir ejecutando líneas de código, tal como se muestra en el siguiente código.

```
$ ipython
Python 3.8.5 (default, Sep 4 2020, 02:22:02)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %pylab
Using matplotlib backend: MacOSX
Populating the interactive namespace from NumPy and matplotlib

In [2]: from numpy import linspace, sin

In [3]: from matplotlib.pyplot import plot

In [4]: x = linspace(-10.,10., 1000) # el punto indica que es de formato float

In [5]: plot(x,sin(x))
Out[5]: [<matplotlib.lines.Line2D at 0x118352550>]
```

El resultado de la ejecución del anterior código se muestra en la Figura 1.1.

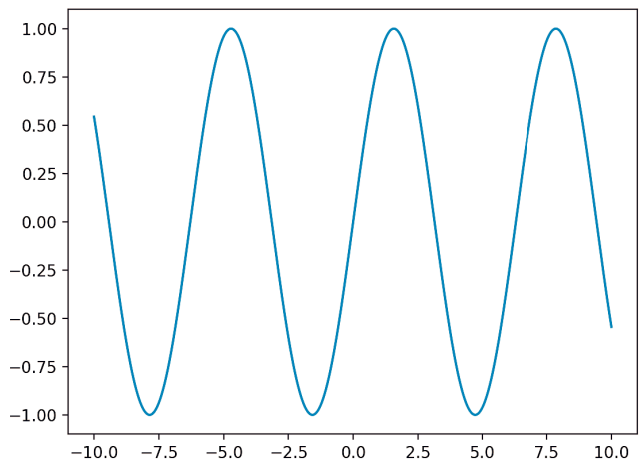

Figura 1.1: Ejemplo de utilización de IPython utilizando la función "mágica" %pylab.

[3] IPython es en realidad una librería que originalmente estaba destinada a mejorar la consola interactiva de Python con el objetivo de hacerla más "amistosa"[Ros18]. Su nombre proviene del carácter interactivo (IPython).

Aparte del aspecto estético, existen múltiples diferencias[4] entre el intérprete de Python y el intérprete mejorado IPhyton. En la dirección https://ipython.readthedocs.io/en/stable/overview.html se ofrece un resumen técnico de las más importantes, un análisis de las cuales se encuentra fuera del objetivo de este texto. En la dirección http://ipython.org/ se pueden encontrar la mayoría de los recursos disponibles e información extensa sobre IPhyton, así como en las referencias [Van16b] y [Joh19].

1.2.2 Material Adicional y Jupyter

En https://github.com/gapsc-us/labcomdig el lector puede encontrar, además de la librería labcomdig.py, los códigos de cada capítulo en versión notebook de Jupyter. Esto es, todos los códigos de cada capítulo están disponible en GitHub en formato .ipybn.

En el caso de Python, los *Notebooks* son ficheros con extensión .ipynb que representan mediante una interfaz gráfica una combinación de código, texto, expresiones matemáticas, figuras, etc. y que fueron introducidos por primera vez en la aplicación Mathematica en 1988. El formato notebook permite ver los códigos junto con sus resultados. Este modelo de *cuaderno computacional* fue adoptado por múltiples aplicaciones y lenguajes y fueron utilizados por primera vez con IPython en 2011 [Ros18]. Posteriormente, en 2014, los desarrolladores de IPython anunciaron el proyecto Jupyter (un nuevo acrónimo) que mejoraba y ampliaba la interfaz Notebooks para Python (no solo para Python) y hoy se ha convertido en un ecosistema por sí mismo.

En la propia web de GitHub se pueden visualizar los códigos de cada capítulo y los resultados de su ejecución. Si adicionalmente el lector quisiera ejecutarlos, podría hacerlo fácilmente desde un servidor en la nube. Se propone utilizar Google Colab. Para ello, de forma transparente y sin instalación, basta con hacer click en el logo de Colab que aparece al inicio de los notebooks facilitados. Se puede cambiar cualquier línea de código y volver a ejecutar.

Si se quisiera ejecutar en local, el lector puede optar por ir copiando el código en los notebooks y salvarlo como .py en Spyder. Alternativamente, podría instalar Anaconda-Navigator que trae su propio Spyder y el Jupyter Notebook. Una vez arrancado el Jupyter, aparecerá en su navegador web una página con el directorio de su disco duro donde podrá seleccionar el notebook que desee y abrirlo para ejecutarlo.

El proceso de instalación es tedioso y conlleva el uso de environments si se quiere gestionar bien. Estos environments permiten instalar y gestionar los paquetes necesarios, que en este caso son el matplotlib, numpy y scipy. En el Apéndice encontrará más información para llevar a cabo la instalación de Anaconda, lo que tal como se ha comentado, le permitirá utilizar Jupiter en su ordenador personal, para poder ver y ejecutar los códigos de este texto.

[4] Una de las características más interesantes de IPhyton consiste en la posibilidad de utilizar la tecla TAB para autocompletar con nombres válidos lo ya tecleado. Esta compleción es contextual. Asimismo, tecleando ? tras un comando o función, se accede a la ayuda contextual.

1.3 Características básicas de Python

En este apartado se realizará un breve resumen de las principales características del lenguaje Python, utilizándose el intérprete `IPython`. Una importante referencia para este apartado se encuentra en el texto de Lutz [Lut13] y en los libros de Jake VanderPlas, [Van16b] y [Van16a]. Este último presenta de una manera simple y sencilla un interesante recorrido por el lenguaje Python y en la dirección https://github.com/jakevdp/WhirlwindTourOfPython/ se puede encontrar el código utilizado en el libro y el uso de los mencionados Notebooks para ilustrarlo. También es posible descargar los Notebooks correspondientes a [Van16b] en la dirección https://github.com/jakevdp/PythonDataScienceHandbook/tree/master/notebooks.

1.3.1 Aspectos básicos del lenguaje Python

En un rápido y breve repaso de los aspectos básicos del lenguaje se puede empezar por explicar algunas reglas sintácticas a la hora de escribir un programa en Python. Es decir, cómo es la estructura de un programa escrito en Python.

La primera regla que se puede señalar es que siempre que una línea comience con el signo `#` se interpreta como un comentario. También todo lo que haya a la derecha de un `#` no se ejecuta, al considerarse un comentario. Y se puede comentar un grupo de líneas mediante tres apóstrofes `'''` en la línea inicial y otros tres apóstrofes `'''` al final del bloque de comentarios.

Con diferencia con otros lenguajes interpretados, Python no tiene un carácter que señale el fin de línea, diferente al carácter invisible del propio fin de línea. Si se desea que una sentencia continue en una segunda línea, por razones estéticas o por alguna otra causa, es posible indicarlo haciendo uso del carácter `\` aunque también es posible prescindir de este carácter en determinados casos. Las dos expresiones siguientes serían equivalentes:

```
In [1]: x = 1+2+3+4+5+\
   ...: 6+7+8

In [2]: y = (1+2+3+4+5+
   ...: 6+7+8)
```

La secuencia de caracteres `...:` se produce automáticamente por el intérprete `IPython`, de manera equivalente a las expresiones `In [x]:` ó `Out[x]:`. La mayoría de las guías de estilo de Python aconsejan la segunda de las formas, con paréntesis, frente a la primera. De igual manera, se desaconseja por las guías el uso del carácter `;` para continuar en una misma línea de texto con dos o más sentencias, como se hace en la siguiente expresión:

```
In [1]: z = x+y; w = x-y
```

prefiriéndose esta otra:

```
In [1]: z = x+y
In [2]: w = x-y
```

En Python los espacios en blanco importan. Esencialmente para señalar un bloque de código que deberá ser tratado como una unidad, en el que el bloque se identifica mediante una indentación, bloque que siempre está precedido por los dos puntos, `:`, como puede observarse en las siguientes líneas (el significado de las mismas se estudiará más adelante):

```
1  In [1]: total = 0
2
3  In [2]: for i in range(100):
4      ...:    total += i
5      ...:
6  In [3]: print(total)
7  4950
```

En particular, el fin de un bucle `for` o `while` o un condicional como `if`, utilizado en el ejemplo anterior, no se indica con el habitual `end`, sino terminando el indentado, de forma que la falta de indentación de la línea `6` es fundamental. En el código siguiente el resultado, no mostrado, sería completamente diferente.

```
1  In [1]: total=0
2
3  In [2]: for i in range(100):
4      ...:    total += i
5      ...:    print(total)
6      ...:
```

Sin embargo, dentro de una línea *los espacios en blanco no importan*. A este respecto, cabe señalar que la mayoría de las guías de estilo de Python recomiendan el uso de un espacio en blanco alrededor de los operadores binarios, tal como se ha hecho en los códigos anteriores alrededor de los operadores `+` y `=`.

Hay que señalar que en Python se distinguen las mayúsculas de las minúsculas. Por ejemplo, `transmisor`, `Transmisor`, `TransMisor`, serían tres entidades diferentes.

Para finalizar este breve repaso de los aspectos sintácticos de Python, señalar que se ha mencionado la existencia de *guías de estilo*. La utilizada más ampliamente (es solo una recomendación para la escritura de código Python) se conoce como PEP8 y puede encontrarse en la dirección https://www.python.org/dev/peps/pep-0008/.

1.3.2 Objetos, atributos y métodos

El paradigma de programación más extendido para la escritura de programas de Python es el *orientado a objeto*. En este sentido, todo en Python es un objeto, incluido el propio programa de Python o los módulos, paquetes y librerías escritos en Python.

No corresponde a este manual explicar qué es una programación orientada a objeto, pero será necesario detenerse en aclarar algunos conceptos que facilitarán la compresión y escritura del código.

Muy esquemáticamente, y probablemente de una manera poco ortodoxa, se puede decir que un objeto es un ejemplo (se debería utilizar la palabra más precisa *instancia*) de elemento que pertenece a una determinada *clase*, que posee ciertos *atributos*, propiedades asociadas a la instancia con un determinado valor, y sobre el que se pueden realizar ciertas operaciones o *métodos*.

Se puede pensar, por ejemplo, en el número `5`. Es un *ejemplo (instancia)* de la *clase* de los números enteros, posee determinados *atributos* o propiedades, por ser un número entero. También se pueden realizar operaciones sobre el mismo, conocidas como *métodos*, que conceptualmente pueden entenderse como una especie de funciones asociadas a la clase a la que pertenece el objeto. Cada clase tiene definida sus propios métodos que no tienen que ser compartidos con otra clase. Por ejemplo, se pueden sumar dos instancias de la clase de los números enteros. Sin embargo, no están definidos otros métodos: no es posible concatenar dos números enteros, operación que si se puede hacer con otras instancias de otras clases de objetos, de la clase `str`, por ejemplo.

Se accede a los atributos y métodos de los objetos mediante la sintaxis del punto `(.)`. Un ejemplo muy simple de esta sintaxis se muestra en las siguientes líneas de código.

```
In [1]: x = 3.25

In [2]: print(x.real, "+" , x.imag, "i")
3.25 + 0.0 i

In [3]: x.as_integer_ratio()
Out[3]: (13, 4)
```

Cuando se ha asignado[5] a la *variable* `x` el valor `3.25`, Python ha creado por defecto un objeto `3.25` de la clase `float`. Este objeto de la clase `float` tiene como *atributos* `real`, `imag`, a los que se accede mediante las instrucciones `x.real` y `x.imag`, que tienen como valores en este caso concreto `3.25` y `0`, respectivamente y como *método*, entre otros, `as_integer_ratio()`, que nos devuelve `(13, 4)`. En este código se ha utilizado la variable `x` para referirse al objeto de la clase `float` `3.25`. También se podría prescindir de la variable y escribir:

```
In [1]: print(3.25.real, "+" , 3.25.imag, "i")
3.25 + 0.0 i

In [2]: 3.25.as_integer_ratio()
Out[2]: (13, 4)
```

[5] En un apartado posterior se aclara el significado de los conceptos *variables* y *objetos*.

Para acceder a los métodos de los diferentes objetos de Python mediante la sintaxis del punto pueden seguirse dos procedimientos con resultados similares:

```
instancia.metodo(args1, args2, ...)
clase.metodo(instancia, args1, args2, ...)
```

El uso de ambos procedimientos puede observarse en el código que se presenta a continuación.

```
In [4]: x.as_integer_ratio()
Out[4]: (13, 4)

In [5]: float.as_integer_ratio(x)
Out[5]: (13, 4)

In [5]: float.as_integer_ratio(3.25)
Out[5]: (13, 4)
```

1.3.3 Variables, objetos y referencias

Las variables y objetos son las principales formas en las que se operan, referencian y almacenan datos dentro de Python.

Una variable se crea en Python cuando se pone un *nombre* o *variable* a la izquierda del signo = y a la derecha un *objeto* conocido. Es decir, cuando se escribe:

```
In [1]: total = 5
```

al menos conceptualmente al interpretar esa línea, Python realiza tres pasos consecutivos, [Lut13]:

1. *Crea* un objeto para almacenar y representar el valor 5. *Crea* el objeto 5.
2. *Crea* la variable `total`, si aún no existe.
3. *Enlaza* la variable `total` al nuevo objeto 5.

Así, cuando en Python se escribe la línea `total=5` se crea el *nombre* `total` que es un *puntero* que apunta a un objeto particular en cada momento, en este caso, el 5, una instancia de una clase definida, y a continuación se *enlaza* la variable con el objeto (instancia). Estos enlaces de las variables a los objetos se denominan en Python *referencias*. En términos más concretos, siguiendo nuevamente a Lutz [Lut13]:

- Las *variables* son entradas (nombres) en una tabla del sistema, con enlaces a objetos.
- Los *objetos* son zonas de memoria con suficiente espacio asignado para representar los valores que poseen y propiedades que los definen.
- Las *referencias* son un tipo de asociación, implementada como un puntero en memoria, entre las variables y los objetos.

Las variables no son los objetos, pero mediante ellas (su nombre) es posible referirse al objeto, a sus atributos y a sus métodos, mediante la sintaxis de punto. Las variables en Python no son más que *nombres* que definen *punteros* a *objetos* de una determinada *clase*. Por ejemplo, si erróneamente se tratara de utilizar una variable que no se ha definido previamente, el intérprete de `IPython` devolvería el siguiente error:

```
In [1]: print(n)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-29-349653be3a92> in <module>
----> 1 print(n)

NameError: name 'n' is not defined
```

Con diferencia a otros lenguajes de programación, en Python no es necesario declarar el tipo al que pertenece una variable porque, en realidad, en Python una variable nunca contiene información acerca de que tipo es. Esto es así porque, en Python, como se ha dicho, se puede considerar que una *variable* es el nombre de un *puntero* que apunta a un objeto en particular en cada momento. Ese objeto sí contiene información del tipo que es, en este caso un *entero*. Por ello, la noción de *tipo* hay que asociarla en Python al objeto y no al nombre o variable que en un momento particular apunta a ese objeto.

Por ejemplo, utilizando nuevamente por adelantado algunas funciones que se definirán posteriormente, se podría ejecutar el siguiente código, con las líneas de resultados que se muestran.

```
In [1]: x = 4

In [2]: type(x)
Out[2]: int

In [3]: x = 'Comunicaciones Digitales'

In [4]: type(x)
Out[4]: str

In [5]: x = [1,2,3]

In [6]: type(x)
Out[6]: list
```

Como consecuencia de que las variables no tienen tipos, con la simple asignación de un objeto a una variable no se cambia el tipo de la variable, que no existe, sino que la variable apunta a un objeto que tiene un tipo diferente.

Hay una consecuencia inmediata de que una variable pueda considerarse en Python un puntero y es el efecto que se produce cuando una variable apunta a un tipo de objetos que se denominan mutables, que se definirá posteriormente. En una de las

líneas de código anteriores se ha definido la variable `x = [1,2,3]`. Escribiendo las siguientes líneas, [Van16a], en el intérprete `IPython`,

```
In [1]: x = [1,2,3]

In [2]: y = x

In [3]: print(y)
[1, 2, 3]

In [4]: x.append(4) #añade 4 a la lista apuntada por x

In [5]: print(y)
[1, 2, 3, 4]
```

se obtiene el resultado mostrado, que puede conducir a confusión si se piensa que una variable en Python es un contenedor de un valor. En efecto, modificando el valor de `x` se ha modificado el valor de `y`. Pero, en realidad, si se piensa en que una variable en Python es un puntero, el resultado es coherente. Cuando se escribe `y = x` en realidad lo que se está haciendo es que la variable `y` apunte al mismo objeto que `x`. Es decir, se están creando dos nombres para el mismo objeto, la lista `[1, 2, 3]`. Este objeto en particular es de un tipo que puede modificarse con el *método* `append`. Por lo tanto, cuando se modifica, puesto que tanto `x` como `y` apuntan al mismo objeto, el *valor que tienen cambia*.

El comportamiento es completamente diferente si a `x` se le asignara otro valor:

```
In [6]: x = [5, 6, 7]

In [7]: print(x)
[5, 6, 7]

In [8]: print(y)
[1, 2, 3, 4]
```

En este caso, la primera línea hace que la variable `x` apunte a un nuevo objeto, en este caso un objeto de tipo `list` y valor `[5, 6, 7]`. Sin embargo, el valor de `y` no ha cambiado puesto que sigue apuntando al mismo objeto anterior.

Este comportamiento puede parecer confuso inicialmente pero es fácil de entender si se piensa en una variable como un puntero, tal como se ha dicho anteriormente. Igualar dos variables equivale a hacer que las dos variables apunten a un mismo objeto. Si no se produce una asignación de un valor diferente a una variable, lo que en realidad es hacer que esa variable apunte a un nuevo objeto, y se cambia el valor del objeto inicialmente apuntado por las dos (o más variables), entonces todas y cada una de ellas *tomarán un valor diferente*[6].

[6] Es necesario repetir que una variable en Python *no tiene un valor*, de ahí que se resalten las anteriores palabras para señalar que representan una forma de hablar.

1.3.4 Tipos de datos: mutables e inmutables

Por defecto, Python contiene unos tipos de datos incorporados[7] (nativos), como la mayoría de lenguajes de programación, tales como `int` para enteros o `float` para decimales. En la Tabla 1.2 se recogen estos tipos.

Tabla 1.2: Tipos de datos nativos de Python.

Tipo/Clase	Qué representan	Ejemplo
str	Cadena de texto, inmutable	'Texto'
list	Secuencia mutable	[3.0, 'Texto', True]
tuple	Secuencia inmutable	(3.0, 'Texto', True)
set	Conjunto mutable, sin orden	set([3.0, 'Texto', True])
frozenset	Conjunto inmutable, sin orden	frozenset([3.0, 'Texto', True])
dict	Diccionario. Clave:valor	{'Edad' : 25, 'Nombre': 'Manuel'}
int	Núm. entero. Precisión fija.	490
float	Núm. decimal. Coma flotante	4.3457821
complex	Núm. complejo. Coma flotante	4.1 + 3.7j
bool	Booleano	True ó False

Se puede observar en la Tabla 1.2 que se señala el carácter *mutable* o *inmutable* de varios de los tipos de datos[8]. Un objeto en Python es *mutable* cuando se puede modificar mediante algunos de los métodos de su clase. Si se toma como ejemplo una instancia de objeto del tipo `list`, en el siguiente código puede entenderse perfectamente el concepto de *mutabilidad*.

```
In [1]: L = ['pam', 'psk', 'qam'] # Una lista de tres datos del tipo str

In [2]: L.append('fsk')

In [3]: L
Out[3]: ['pam', 'psk', 'qam', 'fsk']

In [4]: print(dir(L))
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '
     __dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__'
     , '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__'
     , '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '
     __mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
      '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
     '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', '
     extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

[7] Es decir, *clases de objetos*.

[8] Puede observarse que, en función del contexto, la palabra *dato* se utiliza de manera sinónima a una *instancia* de una clase de objeto. Un *dato* será por tanto una instancia de una clase de objeto.

```
10
11  In [5]: L.reverse()
12
13  In [6]: L
14  Out[6]: ['fsk', 'qam', 'psk', 'pam']
```

En la linea 1 se define una variable L, igual[9] a un objeto del tipo list, como una secuencia de tres objetos (instancias) de tipo str. El objeto de la clase list es, según Tabla 1.2, mutable. En este caso, se puede añadir elementos a la lista mediante uno de los métodos asociados al objeto, el método append(), como se hace en la línea 3. En la línea 6 se observa que L tiene ahora cuatro elementos: el objeto es *mutable*.

En la línea 8 se ha llamado a la función nativa de Python print, con el argumento dir(L). dir() también es una función nativa de Python, que devuelve el *directorio*, la lista, de atributos y métodos asociados a un objeto, en este caso del tipo list. En la línea 11, se hace uso de otros de los métodos definidos y puede verse en la línea 14 como la variable L apunta a un objeto que ha cambiado, invirtiendo los valores de la secuencia, lo que demuestra la mutabilidad del objeto de tipo list.

Una clase de objetos *inmutable* es la clase tuple, que crea una secuencia. En el siguiente código se muestra lo que ocurre cuando se intenta ejecutar un conjunto de instrucciones similares a la anterior. Es interesante observar que un objeto de la clase list se crea, por ejemplo, encerrando entre [] una lista de datos, y un objeto de la clase tuple se crea encerrando la lista entre paréntesis, ().

```
In [1]: T = ('pam', 'psk', 'qam', 'fsk')

In [2]: T.reverse()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-8-2ccbe52649f2> in <module>
----> 1 T.reverse()

AttributeError: 'tuple' object has no attribute 'reverse'
```

Por defecto, Python entiende que aj, con a un número real, tiene valor igual a $a\sqrt{-1}$. Esto permite definir y operar con números complejos, para lo cual se asigna a una variable cualquiera un número complejo, en la forma, por ejemplo, c = 4.32+3.1j. Nótese que para $a = 1$, se tiene 1j, esto es, $\sqrt{-1}$.

Python tienen definidas varias funciones o métodos por defecto para trabajar con números. Como se ha dicho, al igual que en cualquier lenguaje de programación, un método o función en Python es una sentencia que recibe una serie de parámetros y tras aplicar una lógica o funcionalidad interna devuelve otra serie de variables. Algunos métodos (funciones) matemáticas que Python tiene definido por defecto están recogidos en la Tabla 1.3.

[9] Es importante recordar el carácter de *puntero* de una variable (nombre).

Tabla 1.3: Funciones matemáticas comunes de Python.

Función	¿Qué hace?
`max(a,b,c,...)`	devuelve el máximo valor de los proporcionados
`min(a,b,c,...)`	devuelve el mínimo valor de los proporcionados
`abs(num)`	devuelve el valor absoluto del número proporcionado
`len(list or str)`	devuelve número de elementos de una lista o string
`sum(list)`	suma los elementos de una lista
`help(method)`	representa información de ayuda sobre un método

Para trabajar con vectores y matrices, se podría utilizar la clase `list` nativa de Python, o bien crear una clase adecuada para trabajar directamente con vectores y matrices. Sin embargo, este no será el procedimiento habitual seguido en este manual, pues la librería `NumPy` tiene definida clases de objetos adecuadas a este objetivo, tal como se podrá ver más adelante.

1.3.5 Importación de librerías, paquetes y módulos

De acuerdo con la jerarquía conceptual de Python, ver [Lut13], un programa de Python puede descomponerse en módulos, módulos que contienen declaraciones, declaraciones que contienen expresiones y expresiones que *crean y procesan objetos*.

Más específicamente, un programa o script de Python está formado por un fichero `.py` que es interpretado por Python. Este fichero, que se puede imaginar como fichero principal, puede llegar a ser lo suficientemente largo como para resultar intratable, o puede contener una gran cantidad de código que resulte conveniente tener escrita en otros ficheros `.py` para, por ejemplo, ser utilizada por otros ficheros principales. Estos ficheros, que pueden o no ser ejecutables por el intérprete, pueden contener, por ejemplo, definiciones de funciones utilizables en diversas circunstancias.

Este carácter modular de la programación en Python ha permitido la creación de un numeroso grupo de librerías, paquetes y módulos de propósitos especiales y generales accesible para su uso por cualquier programador. En este apartado se estudiará la forma en la que estos elementos se incorporan dentro de un programa escrito en Python y la manera en la que se accede a los elementos que lo constituyen.

Es conveniente detenerse brevemente en la distinción entre librerías, paquetes y módulos. Esta es una estructura jerárquica, en la que un *módulo* es un único fichero `.py`, en general no ejecutable, que contiene definiciones y sentencias. Estas definiciones y sentencias pueden ser *importadas* en otros módulos (reutilizadas) o en un programa principal. Un *paquete* es una forma de organizar y agrupar diferentes módulos con algunas características en común.

Una *librería*, y específicamente la librería standard de Python, de acuerdo con la documentación oficial de Python https://docs.python.org/3/library/intro.html, es un

conjunto de paquetes, módulos y elementos, no necesariamente escritos en Python, que permite acceder a funcionalidades del sistema, funciones específicas, módulos diseñados para aplicaciones específicas, etc. Existen librerías de propósitos especiales que aportan funcionalidades determinadas, tipos de datos diferentes a los incorporados por la librería standard para desarrollos en determinados entornos, etc.

Para el uso de las diversas librerías, excepto la librería standard, que incluye las funciones y clases de objetos básicos de Python, es necesario en primer lugar instalarlas, como se señaló en el Apartado A.2, y a continuación, importarlas. En el caso de un paquete o módulo, solo es preciso la importación. La *importación* consiste esencialmente en hacer accesible mediante un conjunto de instrucciones los diferentes objetos existentes en la librería, paquete o módulo, al programa que lo importa. Por su interés, y como ejemplo de aplicación, a continuación se explica cómo se importa y se utilizan diferentes elementos de la librería `NumPy`, procedimiento que se sigue de manera similar para cualquier otra librería, paquete o módulo.

La sentencia `import`

La importación de una librería puede realizarse haciendo uso de su nombre completo:

```
import numpy
```

Cuando la sentencia `import numpy` se ejecuta, el nombre `numpy` sirve para dos propósitos: por una lado, identifica el nombre del fichero que se está importando pero también se convierte en una variable que identifica al *objeto* librería importada. A continuación se genera una tabla de nombres[10] (variables) de los objetos existentes en la librería importada a los que se accede mediante la notación del punto.

Es decir, con la ejecución del comando `import numpy` se han importado todas las funciones, métodos y clases de objetos definidas en el fichero `NumPy.py` y se han hecho accesible todos los módulos llamados desde el mismo. Todos esos elementos podrán ser utilizados en el fichero de llamada mediante la notación punto (`.`).

Por ejemplo, en la librería `NumPy` existe el método `array()` para crear `ndarray`, una clase de objetos que comparte algunas características con un tipo de objetos nativo de Python denominado `list`. Si se desea utilizar este procedimiento para crear una instancia de esta clase de objetos es necesario ejecutar un código como el que sigue:

```
In [1]: import numpy

In [2]: a = numpy.array([1,2,3])

In [3]: a
Out[3]: array([1, 2, 3])
```

[10] *Namespace* en la terminología de Python. Puede resultar muy interesante comprender cómo se crea esta tabla de nombres del módulo importado. Ver, por ejemplo, [Bud10].

```
In [4]: type(a)
Out[4]: numpy.ndarray
```

La sentencia `from`

En contraste con la importación de una librería mediante la sentencia `import`, existe la posibilidad de importar un *nombre específico* de una librería (en realidad, una función o método) en un script de llamada mediante un comando diferente, la sentencia `import`, como se muestra en el siguiente código:

```
In [1]: from numpy import array

In [2]: a = array([1, 2, 3])

In [3]: type(a)
Out[3]: numpy.ndarray
```

De esta segunda forma se importa únicamente el método `array()` de NumPy y se *copia* la variable (nombre) *array* en el script de llamada, por lo que ya se puede utilizar directamente como una función o método del sistema. Esta forma de importar es útil para funciones y métodos específicos. Sin embargo, puede que al importar de esta forma se sobrescriban las funciones bases de Python u otras funciones. Por ejemplo NumPy tiene una función `sum` y Python por defecto tiene una función con el mismo nombre. Si se ejecuta `from numpy import sum` se sobreescribe la función `sum` propia de Python, lo que puede tener efectos indeseados (tal vez no en este caso).

Aunque se recomienda la primera forma de importar la librería NumPy, se puede hacer de la forma que resulte más cómoda.

La equivalencia y diferencia entre las dos maneras de importar una librería o algunos de los módulos o nombres que contienen, así como los aspectos técnicos más sobresalientes del significado del proceso de importación, más allá de su uso, puede estudiarse en el Capítulo 22 del texto de Lutz [Lut13] y en [Lut13, págs. 718-724].

La extensión `as` para los comandos `import` y `from`

Existe una manera diferente de uso de los comandos `import` y `from` que permite asignar a los objetos importados un nombre diferente en el script de importación. De acuerdo con esta forma, el siguiente código muestra cómo se importaría la librería NumPy asignándole a las funciones y métodos que la forman un nuevo nombre (*nickname*), que suele ser más corto y fácil de recordar:

```
In [1]: import numpy as np

In [2]: a = np.array([1, 2, 3])
```

El "nickname" (alias) elegido para NumPy suele ser `np`. La mayoría de códigos que pueden encontrarse en los repositorios importan la librería NumPy con ese alias.

1.4 Librería NumPy

En este apartado se van a presentar con detalle los nuevos tipos de datos, clases y funciones que incorpora la librería NumPy (un *nombre corto* para Numerical Python) a los datos, clases y funciones nativas de Python. El principal interés de la librería NumPy radica en aportar una interfaz eficiente para almacenar y operar una nueva clase de objetos, ndarray, formado por una agrupación ordenada de un número arbitrario de elementos homogéneos[11].

La clase de objetos ndarray permite definir *arrays*, que forman el núcleo del uso en el campo científico de Python. En este texto, se utilizará el término *array* para definir cualquier conjunto de estructura de datos y juegan un papel fundamental en Python. El término engloba a los vectores y matrices pero también a listas y tensores. Un atributo clave de los mismos es el número de dimensiones (ejes) que presentan y así, por ejemplo, los vectores y matrices *clásicos* estarán definidos por arrays de dos dimensiones.

Además de la clase de objetos ndarray, NumPy define un amplio conjunto de operaciones matemáticas sobre las instancias de la clase, así como herramientas adecuadas para álgebra lineal, transformada de Fourier, generación de números aleatorios, etc.

Para utilizar la librería NumPy es necesario en primer lugar, como ya se ha dicho, instalarla. A continuación será necesario importarla dentro del programa principal, con alguno de los métodos ya mencionados. Por ello, se puede observar en la mayor parte del código que sigue que las primeras líneas corresponderán a:

```
In [1]: import numpy as np

In [2]: np.__version__
Out[2]: '1.18.1'
```

donde también se ha utilizado el atributo `__.version.__` del objeto np para conocer la versión importada de la librería. Una documentación detallada, junto con tutoriales y otros recursos acerca de NumPy puede encontrarse en http://www.numpy.org.

1.4.1 La clase ndarray de NumPy: vectores, matrices y arrays

El núcleo de NumPy se construye en torno a una clase de objetos inmutables formada por una colección de datos homogéneos (de una misma clase) con una determinada agrupación, denominada ndarray. Una lista completa de los atributos y métodos de la clase puede obtenerse mediante el procedimiento habitual `dir(np.ndarray)`. En la Tabla 1.4 se presentan los atributos básicos de la misma.

Antes de mostrar cómo definir arrays y utilizar los atributos de los mismos, conviene detenerse en las diferencias y similitudes existentes entre los arrays y los vectores

[11] En contraposición con la clase de objeto nativa de Python, list, que puede estar formado por elementos de diferentes tipos.

Tabla 1.4: Atributos básicos de la clase `ndarray`.

Atributo	Descripción
`ndim`	Número de dimensiones (ejes) del array
`shape`	Un secuencia (tuple) que contiene el número de elementos por cada dimensión (eje) del array
`size`	El número total de elementos del array
`nbytes`	Número de bytes utilizado para almacenar los datos del array
`dtype`	El tipo de dato de los elementos del array

y matrices definidos en la literatura científica general. La clase `ndarray` define un objeto formado por agrupaciones de elementos de un único tipo en un conjunto de ejes o dimensiones. Así, una instancia de la clase podrá tener $(1, 2, 3, \cdots)$ *ejes*, (axis) que suelen identificarse en muchas de las funciones definidas en la librería `NumPy` mediante las expresiones `axis=0`, `axis=1`, `axis=2` y así sucesivamente, correspondiendo a las direcciones vertical, horizontal y profundo, respectivamente. En el siguiente código se define una variable `A` a la que se le asigna una instancia de la clase `ndarray`.

```
In [1]: import numpy as np

In [2]: A = np.array([1, 2, 3, 4, 5])

In [3]: A
Out[3]: array([1, 2, 3, 4, 5])
```

Aunque no es formalmente correcto, se puede afirmar que el anterior código crea una variable `A` de la clase `ndarray` con una única dimensión (eje), que en este caso siempre es el eje vertical (`axis = 0`). Si se utilizan los atributos de la clase definidos en la Tabla 1.4, se obtendría lo siguiente:

```
In [4]: A.ndim
Out[4]: 1

In [5]: A.shape
Out[5]: (5,)
```

Es decir, `A` es una variable que apunta a una instancia de la clase `ndarray`, que está definida en un eje vertical, y tiene cinco elementos en ese eje. *No es posible considerar que* `A` *sea un vector fila habitual*, puesto que los vectores y matrices son en realidad arrays que están definidos en los ejes vertical y horizontal. Es importante resaltar que los *vectores* tendrán dos dimensiones, aunque tengan una sola fila (*vector fila*) o una sola columna (*vector columna*). Para construir un vector fila (eje vertical y horizontal, 1×5), con los mismos elementos, se tendría que utilizar la función `np.array()` en una forma diferente:

```
In [9]: A = np.array([[1,2,3,4,5]])

In [10]: A
Out[10]: array([[1, 2, 3, 4, 5]])

In [11]: A.ndim
Out[11]: 2

In [12]: A.shape
Out[12]: (5,1)
```

Puede observarse la diferencia entre las líneas de salida `Out[3]: array([1, 2, 3, 4, 5])` y `Out[10]: array([[1, 2, 3, 4, 5]])`.

Más adelante se presentarán diversas funciones y métodos para operar sobre las dimensiones de los array. Por ejemplo, se puede añadir un eje a un array con la función `np.newaxis`, como se describe en el siguiente ejemplo.

```
In [1]: import numpy as np

In [2]: A = np.array([1, 2, 3, 4, 5])

In [3]: A.ndim
Out[3]: 1

In [4]: Afila = A[np.newaxis,:]

In [5]: Afila.shape
Out[5]: (1, 5)

In [6]: Afila.ndim
Out[6]: 2

In [7]: Acol = A[:,np.newaxis]

In [8]: Acol.shape
Out[8]: (5, 1)

In [9]: Acol.ndim
Out[9]: 2

In [10]: A
Out[10]: array([1, 2, 3, 4, 5])

In [11]: Afila
Out[11]: array([[1, 2, 3, 4, 5]])

In [12]: Acol
Out[12]:
array([[1],
       [2],
       [3],
```

```
       [4],
       [5]])
```

En el Código 1.1 se recogen otros ejemplos de construcción de arrays y del uso de los atributos de los mismos. Es importante recordar la sintaxis del punto (`.`) y cómo es posible acceder a los atributos de una instancia de una clase de dos formas diferentes.

Código 1.1: Construcción de `ndarray`. Atributos.

```
In [1]: import numpy as np

In [2]: A = np.array([[1, 2], [3, 4], [5, 6]])

In [4]: A
Out[4]:
array([[1, 2],
       [3, 4],
       [5, 6]])
In [5]: A.shape
Out[5]: (3, 2)

In [6]: np.shape(A)
Out[6]: (3, 2)

In [7]: A.size
Out[7]: 6

In [8]: np.ndim(A)
Out[8]: 2

In [9]: B = np.array([1, 2, 3, 4, 5, 6])

In [10]: B.ndim
Out[10]: 1

In [11]: B.shape
Out[11]: (6,)

In [12]: C = np.array([[1, 2, 3, 4, 5, 6]])

In [13]: C.ndim
Out[13]: 2

In [14]: C.shape
Out[14]: (1, 6)

In [15]: A.nbytes
Out[15]: 48

In [17]: C.nbytes
Out[17]: 48
```

```
In [18]: C.dtype
Out[18]: dtype('int64')
```

1.4.2 Tipos de datos en NumPy

En el último ejemplo se ha podido observar que el atributo `dtype` de los objetos utilizados era `int64`, un tipo de dato no nativo de Python. En la Tabla 1.5, [Joh19], se resumen los datos numéricos básicos de NumPy.

Tabla 1.5: Tipos de datos numéricos básicos de NumPy.

dtype	Variantes	Descripción
int	int8, int16, int32, int64	Enteros
uint	uint8, uint16, uint32, uint64	Enteros sin signo (no negativos)
bool	Booleano	Toma valor True o False
float	float16, float32, float64, float128	Números de punto flotante
complex	complex64, complex128, complex256	Números complejos de punto flotante

1.4.3 Funciones en NumPy

La librería NumPy tiene predefinida una gran cantidad de funciones matemáticas, entre las que se pueden destacar las recogidas en la Tabla 1.6.

Tabla 1.6: Funciones matemáticas comunes de NumPy.

Función	¿Qué hace?
`np.ceil(x)`	redondea hacia el menor entero mayor que `x`
`np.floor(x)`	redondea hacia el mayor entero menor que `x`
`np.round(x)`	redondea hacia el entero más próximo a `x`
`np.sqrt(x)`	raíz cuadrada `x`
`np.sin(x)`	seno de `x` (`x` medido en radianes)
`np.abs(x)`	valor absoluto o módulo de un número complejo
`np.conj(x)`	conjugado del número complejo `x`
`np.exp(x)`	número e elevado a `x`
`np.log(x)`	logaritmo neperiano de `x`
`np.log2(x)`	logaritmo en base 2 de `x`
`np.log10(x)`	logaritmo en base 10 de `x`

1.5 Construcción, direccionamiento y conformación de arrays

En este apartado se va a profundizar en la generación, direccionamiento y diversas operaciones sobre los objetos de la clase `ndarray`, haciendo uso de los recursos de la librería.

1.5.1 Construcción de arrays

La forma básica de construir arrays en Python consiste en el uso de la función `np.array`, introduciendo sus elementos entres corchetes, tal como se vio en el Código 1.1. Los corchetes externos agrupan filas y los corchetes internos definen cada fila. Se sugiere que se ejecuten las sentencias recogidas en el Código 1.1 y se hagan pruebas similares para comprobar la funcionalidad de los comandos presentados. Por ejemplo, se puede observar que se separan los elementos dentro de una fila mediante comas (`,`) dentro de corchetes y las filas se separan igualmente mediante comas.

A la hora de definir un array en `NumPy` hay que tener en cuenta cómo se va a trabajar con él, ya que se pueden definir arrays de una dimensión (1-D) siempre que las operaciones que se vayan a realizar con ellos sean elemento a elemento, es decir, cuando se quiera aplicar la misma operación por igual a un conjunto de elementos. En cambio, si se desea realizar operaciones con matrices tales como multiplicación de una matriz por un vector, resolución de ecuaciones lineales, producto escalar de vectores, etc., habrá que definir arrays de dos dimensiones (2-D).

Las siguientes líneas de código muestran una vez más la diferencia entre los arrays de una, dos o más dimensiones, utilizando los atributos básicos definidos en la Tabla 1.4.

```
In [1]: import numpy as np

In [2]: A = np.array([1,2,3]) # Definición de un array de una dimensión (1-D).

In [3]: print("Nº de dimensiones de A:", A.ndim, ", Shape de A:", A.shape)
Nº de dimensiones de A: 1 , Shape de A: (3, )

In[4]: B = np.array([[1,2,3]])  # Definición de un array dos dimensiones (2-D) (1,
     3) (vector fila). Obsérvese el doble corchete.

In [5]: print("Nº de dimensiones de B:", B.ndim, ", Shape de B:", B.shape)
Nº de dimensiones de B: 2 , Shape de B: (1, 3)

In[6]: C = np.array([[1],[2],[3]])  # Definición de un array dos dimensiones (2-D)
      (3,1) (vector columna).

In [7]: print("Nº de dimensiones de C:", C.ndim, ", Shape de C:", C.shape)
Nº de dimensiones de C: 2 , Shape de C: (3,1)

In [8]: D = np.array([[[0], [1], [2]]])

In [9]: print("Nº de dimensiones de D:", C.ndim, ", Shape de D:", D.shape)
Nº de dimensiones de D: 3 , Shape de D: (1, 3, 1)
```

En la Tabla 1.7 se resumen algunos de las funciones más habituales para construir arrays. A continuación de la misma se muestran ejemplos de la utilización de algunas de ellas.

Tabla 1.7: Funciones de `NumPy` para la generación de arrays.

Comando	Resultado
`np.array(a)`	Crea un array a partir del argumento `a`, que puede ser un objeto tipo `list` de Python, listas anidadas u otra instancia `ndarray`.
`np.arange(a, b)`	Crea un array de 1-D, con elemento inicial `a` y final `b` de uno en uno, *excluyendo el elemento final* `b`.
`np.arange(a, b, c)`	Crea un array 1-D, con elemento inicial `a` y final `b` de `c` en `c`, *excluyendo el elemento final* `b`.
`np.linspace(a, b, c)`	Genera un array 1-D de `c` elementos linealmente espaciado entre `a` y `b` incluyendo ambos dos. Si el parámetro `c` no está presente, genera 50 elementos.
`np.logspace(a, b, c)`	Genera un array 1-D de `c` elementos logarítmicamente espaciados entre 10^a y 10^b. Si el parámetro `c` no está presente, genera 50 elementos.
`np.zeros([m,n])`	Crea un array de ceros con las dimensiones especificadas. El valor `n` podría dejarse en blanco, generándose un array 1-D.
`np.ones([m,n])`	Crea un array de unos con las dimensiones especificadas. El valor `n` podría dejarse en blanco, generándose un array 1-D.
`np.diag([v][,k=])`	Crea un array, de las dimensiones especificadas, con los valores de `v` en la diagonal principal, si el parámetro opcional `k` es cero (valor por defecto). Si `k>0` coloca `v` en la diagonal `k` por encima de la principal y si `k<0` coloca `v` en la diagonal `|k|` por debajo de la principal.
`np.eye(m,n)`	Crea un array, con las dimensiones especificadas, de unos en la diagonal principal y ceros en el resto.

(...)

Tabla 1.7: (continuación).

Comando	Resultado
`np.random.rand(m,n)`	Crea un array, con las dimensiones especificadas, de números aleatorios con distribución uniforme entre [0,1].
`np.random.randint(IMIN,IMAX, [m,n])`	Crea un array, con las dimensiones especificadas, de números enteros con distribución discreta uniforme entre [`IMIN : IMAX`) (*excluyendo* `IMAX`).
`np.random.randn(m,n)`	Crea un array con las dimensiones especificadas de números aleatorios con distribución normal $\mathcal{N}(0,1)$.
`np.kron(x,y)`	Realiza el producto de Kronecker, sustituyendo cada entrada del array `x` por el producto de la misma por el array `y` completo.

```
In [1]: import numpy as np

In [2]: x = np.arange(9)        #Un array 1-D formado por números comprendidos
     entre 0 y 9, excluyendo el 9

In [3]: x
Out[3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8])

In [4]: y = np.arange(1, 8, 2)  # Un array 1-D formado por números comprendidos
     entre 1 y 8 de dos en dos
```

Se pueden crear arrays formados con secuencias incrementadas lineal o logarítmicamente. Estos últimos admiten un parámetro adicional, la base del logaritmo, que por defecto es 10:

```
In [23]: np.linspace(0, 10, 20)
Out[23]:
array([ 0.      ,  0.52631579,  1.05263158,  1.57894737,  2.10526316,
        2.63157895,  3.15789474,  3.68421053,  4.21052632,  4.73684211,
        5.26315789,  5.78947368,  6.31578947,  6.84210526,  7.36842105,
        7.89473684,  8.42105263,  8.94736842,  9.47368421, 10.        ])

In [24]: np.linspace(0, 10, 10)
Out[24]:
array([ 0.        ,  1.11111111,  2.22222222,  3.33333333,  4.44444444,
        5.55555556,  6.66666667,  7.77777778,  8.88888889, 10.        ])

In [25]: np.logspace(0, 2, 8)
Out[25]:
```

```
array([ 1.      ,  1.93069773, 3.72759372, 7.19685673,
       13.89495494, 26.82695795, 51.79474679, 100.  ])

In [26]: np.logspace(0,8,10, base=2)
Out[26]:
array([ 1.      ,  1.85174942, 3.42897593, 6.34960421,
       11.75787594, 21.77264 , 40.3174736 , 74.65785853,
      138.24764658, 256.    ])
```

Para evitar confusiones con el orden del paso de parámetros a funciones, en Python se permite llamar a una función pasándole los argumentos que recibe como pares de `clave=valor`, como se ha podido ver en el último ejemplo de código anterior, en la forma `base=2`, ó en el siguiente ejemplo, donde se especifica el comienzo o el final com las claves `start` o `stop` para `np.linspace` ó `step` para señalar la separación entre los valores:

```
In [1]: import numpy as np

In [2]: x = np.arange(start=1,stop=9)

In [3]: x
Out[3]: array([1, 2, 3, 4, 5, 6, 7, 8])

In [4]: y = np.arange(start=1,stop=8,step=2)  #Un array 1-D formado por números
     comprendidos entre 1 y 8 de dos en dos
In [5]: y
Out[5]: array([1, 3, 5, 7])
```

Generación de arrays constantes

La creación de arrays con valores constantes se puede realizar mediante las funciones utilizadas en los siguientes ejemplos. Es importante señalar la diferencia en los parámetros de llamada de la función `np.eye` frente al resto de funciones.

```
In [1]: import numpy as np

In [2]: np.zeros([2,3])
Out[2]:
array([[0., 0., 0.],
       [0., 0., 0.]])

In [3]: np.ones(4)
Out[3]: array([1., 1., 1., 1.])

In [4]: np.ones(4).shape
Out[4]: (4,)

In [5]: np.ones([4,2])
Out[5]:
array([[1., 1.],
```

```
      [1., 1.],
      [1., 1.],
      [1., 1.]])

In [6]: np.ones([4,2], dtype=np.complex64)
Out[6]:
array([[1.+0.j, 1.+0.j],
      [1.+0.j, 1.+0.j],
      [1.+0.j, 1.+0.j],
      [1.+0.j, 1.+0.j]], dtype=complex64)

In [7]: np.ones([4,2], dtype=np.complex64).dtype
Out[7]: dtype('complex64')

In [8]: np.diag([1,2,3,4])
Out[8]:
array([[1, 0, 0, 0],
      [0, 2, 0, 0],
      [0, 0, 3, 0],
      [0, 0, 0, 4]])

In [9]: np.diag([1,2,3,4],k=2)
Out[9]:
array([[0, 0, 1, 0, 0, 0],
      [0, 0, 0, 2, 0, 0],
      [0, 0, 0, 0, 3, 0],
      [0, 0, 0, 0, 0, 4],
      [0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0]])

In [10]: np.diag([1,2,3,4],k=-1)
Out[10]:
array([[0, 0, 0, 0, 0],
      [1, 0, 0, 0, 0],
      [0, 2, 0, 0, 0],
      [0, 0, 3, 0, 0],
      [0, 0, 0, 4, 0]])

In [11]: np.eye(3,4)
Out[11]:
array([[1., 0., 0., 0.],
      [0., 1., 0., 0.],
      [0., 0., 1., 0.]])
```

Generación de arrays de números aleatorios

La librería NumPy contiene un conjunto de funciones que permite la generación de arrays formados por muestras de una variable aleatoria (escalar o vectorial), continuas o discretas, con funciones densidad de probabilidad determinada. El siguiente capítulo estará dedicado al estudio de las variables aleatorias (v.a.) pero en éste se muestra un uso elemental de dichas funciones de generación con las siguientes líneas de código.

```
In [38]: np.random.rand(1, 10)  #Genera un array 2-D (1,10) de valores de una v.a.
      uniformemente distribuida entre [0, 1]
Out[38]:
array([[0.48101203, 0.05111162, 0.54953869, 0.37158162, 0.84512292,
       0.98817288, 0.36901814, 0.921353 , 0.52350267, 0.50759903]])

In [40]: np.random.randint(-5,5, [1,10])  #Genera un array 2-D (1,10) de valores
     de una v.a. discreta uniformemente distribuida entre [-5, 5)
Out[40]: array([[ 3, -4, -3, -5, 2, -3, -4, 3, 2, -5]])

In [41]: np.random.randint(-5,5, [1,10]).shape
Out[41]: (1, 10)

In [43]: 2*np.random.randn(1,10)+1  #Genera un array 2-D (1,10) de valores de una
     v.a. gaussiana de media 1 y varianza 4
Out[43]:
array([[ 2.64098639, 1.46689387, 3.44536843, -0.52783056, -0.54753785,
        1.07747099, -1.96044404, -1.33481227, 0.99675058, 2.30565064]])
```

Utilización del operador `np.kron`

Para finalizar este apartado de construcción de arrays, en las siguientes líneas de código se muestra un ejemplo del funcionamiento del operador `np.kron`, que realiza el producto de Kronecker entre dos matrices **A** y **B**, habitualmente representado por $\mathbf{A} \otimes \mathbf{B}$:

```
In [1]: import numpy as np

In [2]: A = np.array([[1,0,2,0,3]])

In [3]: B = np.array([[1,2,3]])

In [4]: np.kron(A,B)
Out[5]: array([[1, 2, 3, 0, 0, 0, 2, 4, 6, 0, 0, 0, 3, 6, 9]])
```

1.5.2 Concatenación y separación de arrays

Además de las funciones anteriores, la librería NumPy suministra un conjunto de funciones que permiten concatenar arrays y separar un array en partes del mismo. En el uso de estas funciones, recogidas en la Tabla 1.8, es importante conocer las dimensiones de los elementos que se desea concatenar y tener en cuenta que si se intentan concatenar dos arrays `a` y `b`, de dimensiones adecuadas, introduciéndolos entre corchetes `[]`, no se obtendrá un array formado por la unión de los dos, sino un objeto del tipo `list`. Por ello es necesario utilizar los métodos de la librería NumPy `concatenate`, `r_.` y `c_.`, como se observa a continuación.

Tabla 1.8: Funciones de NumPy para concatenar y separar arrays.

Función	¿Qué hace?
`np.concatenate([x,y,...])`	Genera un array concatenando los arrays `x`, `y` y los que vengan a continuación. Los arrays se concatenan en la primera dimensión compatible.
`np.concatenate([x,y,...],axis)`	Genera un array concatenando los arrays `x`, `y` y los que vengan a continuación. Los arrays se concatenan en la dimensión especificada por `axis`, si es compatible.
`np.append(x,y, axis)`	Produce un array añadiendo `y` tras `x` en las dimensión expecificada por `axis`. Ambos arrays deben tener las dimensiones adecuadas. Si no se especifica `axis`, los datos se añaden aplanando las dimensiones.
`np.r_[x,y,...]`	Produce un array concatenando las filas de los arrays `x`, `y` y los que vengan a continuación. Ambos arrays deben tener el mismo número de columnas.
`np.c_[x,y,...]`	Produce un array concatenando las columnas de los arrays `x`, `y` y los que vengan a continuación. Ambos arrays deben tener el mismo número de filas.
`np.?stack([x,y])`	Dependiendo de que `?` sea `v`, `h` ó `d`, apila los arrays `x` e `y` utilizando los ejes vertical, horizontal ó profundo, respectivamente.
`np.split(x, indice)`	Divide el array `x` dependiendo del significado de `indice`. Las funciones `np.vsplit()`, `np.hsplit()` y `np.dsplit()` son similares.

```
In [1]: import numpy as np

In [2]: a = np.array([[1, 2]]); b = np.array([[3, 4]]); c = np.array([[5],[6]])

In [3]: np.concatenate([a,b])
Out[3]:
array([[1, 2],
       [3, 4]])
```

```
In [4]: # np.concatenate([a,c]) #error because of dimensions

In [5]: np.concatenate([b,a,b])
Out[5]:
array([[3, 4],
       [1, 2],
       [3, 4]])

In [6]: d = np.concatenate([a, b])

In [7]: d
Out[7]:
array([[1, 2],
       [3, 4]])

In [8]: e = np.concatenate([b, a])

In [9]: e
Out[9]:
array([[3, 4],
       [1, 2]])

In [10]: np.concatenate([d,e])
Out[10]:
array([[1, 2],
       [3, 4],
       [3, 4],
       [1, 2]])

In [11]: np.concatenate([d,e],axis=1)
Out[11]:
array([[1, 2, 3, 4],
       [3, 4, 1, 2]])

In [12]: np.append(d,e,axis=0)
Out[12]:
array([[1, 2],
       [3, 4],
       [3, 4],
       [1, 2]])

In [13]: np.append(d,e)
Out[13]: array([1, 2, 3, 4, 3, 4, 1, 2])

In [14]: np.r_[b,a,b]
Out[14]:
array([[3, 4],
       [1, 2],
       [3, 4]])

In [15]: np.c_[d,c]
Out[15]:
array([[1, 2, 5],
```

```
      [3, 4, 6]])

In [16]: np.c_[d,e]
Out[16]:
array([[1, 2, 3, 4],
      [3, 4, 1, 2]])
```

A continuación se muestran ejemplos de las funciones que permiten apilar arrays. Los arrays que se apilan deben poseer las dimensiones adecuadas y el resultado de la operación es muy similar a la concatenación de arrays, excepto que es posible utilizar las funciones con arrays de dimensiones mezcladas.

```
In [28]: x = np.array([1,2,3])

In [29]: x.ndim
Out[29]: 1

In [30]: A = np.array([[9,8,7],[4,5,6]])

In [31]: A.ndim
Out[31]: 2

In [32]: A.shape
Out[32]: (2, 3)

In [33]: np.vstack([x,A]) # Apilar verticalmente
Out[33]:
array([[1, 2, 3],
      [9, 8, 7],
      [4, 5, 6]])

In [34]: np.vstack([A,x])
Out[34]:
array([[9, 8, 7],
      [4, 5, 6],
      [1, 2, 3]])

In [35]: y = np.array([[0],[0]])

In [36]: y.ndim
Out[36]: 2

In [37]: y.shape
Out[37]: (2, 1)

In [38]: np.hstack([y,A]) # Apilar horizontalmente
Out[38]:
array([[0, 9, 8, 7],
      [0, 4, 5, 6]])

In [39]: np.hstack([A,y])
Out[39]:
```

```
array([[9, 8, 7, 0],
       [4, 5, 6, 0]])
```

Para finalizar este apartado, a continuación se muestran ejemplos de uso de las funciones de NumPy que permiten obtener un grupo de arrays separando (dividiendo) un array que se suministra como argumento, de acuerdo con un índice de puntos de separación.

```
In [41]: A = np.random.randint(1,50, size=[6,6]) # Generación de una matriz de
     enteros aleatorios siguiendo una distribución uniforme entre [1,50) (no
     incluye el 50), de tamaño 6x6

In [42]: A
Out[42]:
array([[16, 29, 28, 44, 23, 32],
       [21, 46, 22, 9, 29, 29],
       [32, 48, 11, 1, 8, 46],
       [ 3, 39, 8, 32, 37, 26],
       [39, 28, 5, 38, 44, 3],
       [19, 39, 23, 40, 47, 42]])

In [48]: A1, A2, A3 = np.split(A, [3,5]) # Separa verticalmente (a lo largo del
     axis = 0) el array A mediante los índices 3 y 5

In [49]: A1
Out[49]:
array([[16, 29, 28, 44, 23, 32],
       [21, 46, 22, 9, 29, 29],
       [32, 48, 11, 1, 8, 46]])

In [50]: A2
Out[50]:
array([[ 3, 39, 8, 32, 37, 26],
       [39, 28, 5, 38, 44, 3]])

In [51]: A3
Out[51]: array([[19, 39, 23, 40, 47, 42]])

In [52]: A1, Am, Ar = np.hsplit(A, [1,4])   # Separa horizontalmente (a lo largo
     del axis = 1) el array A mediante los índices 1 y 4

In [53]: A1
Out[53]:
array([[16],
       [21],
       [32],
       [ 3],
       [39],
       [19]])

In [54]: Am
Out[54]:
```

```
array([[29, 28, 44],
       [46, 22, 9],
       [48, 11, 1],
       [39, 8, 32],
       [28, 5, 38],
       [39, 23, 40]])

In [55]: Ar
Out[55]:
array([[23, 32],
       [29, 29],
       [ 8, 46],
       [37, 26],
       [44, 3],
       [47, 42]])
```

1.5.3 Direccionamiento de los elementos de un array de una dimensión

Para acceder o modificar los elementos de un array, se utilizarán los subíndices correspondientes. Se revisan en este apartado procedimientos específicos para un array 1-D, que se extienden a los arrays de dimensiones superiores sin más que considerar que estos procedimientos se pueden seguir para cada uno de los ejes.

En Python se toma el `0` como primer índice de los elementos que componen un array, y los enteros negativos se utilizan para indexar los elementos desde el fin del array. Así, si `d` es un array 1-D, `d[0]` hace referencia al primer elemento del array y `d[-1]` al último, `d[-2]` al penúltimo y así sucesivamente.

A lo largo de cada eje (o del único eje en el caso de arrays 1-D) se utiliza la notación `:` para especificar un rango de elementos que pueden ser seleccionados con expresiones tales como `m:n` (tanto `m` como `n` ó `p` en lo que sigue deben ser números enteros), que selecciona los elementos de un array que empiezan en el índice `m` y finalizan en el elemento que está en la posición `n-1`; esto es, *el elemento en la posición* `n` *no se incluye*. También pueden seleccionarse elementos utilizando expresiones del tipo `m:n:p`, para seleccionar elementos desde el índice `m` hasta el índice `n` de `p` posiciones en `p` posiciones. Si `p` es negativo, los elementos son devueltos en orden inverso. En este caso, `m` debe ser mayor que `n`. En la Tabla 1.9, [Joh19], se resumen todas estas operaciones y a continuación se muestran ejemplos de su uso.

```
In [1]: import numpy as np

In [2]: a = np.arange(10)

In [3]: a.shape
Out[3]: (10,)

In [4]: a.ndim
Out[4]: 1

In [5]: a[0]
```

Tabla 1.9: Resumen de expresiones para seleccionar elementos de un array 1-D.

Función	¿Qué hace?
`a[m]`	Selecciona el elemento de índice `m`, siendo `m` un entero.
`a[-m]`	Selecciona el elemento de índice `-m`, comenzando por `-1`, que hace referencia al último elemento del array.
`a[m:n]`	Selecciona los elementos que comienzan con en el índice `m` y finalizan con el elemento `n-1`.
`a[:]`	Selecciona todos los elementos.
`a[:n]`	Selecciona los elementos que empiezan en el índice `0` hasta el elemento de índice `n-1`.
`a[m:]` ó `a[m:-1]`	Selecciona los elementos que empiezan en el índice `m` hasta el último elemento del array 1-D.
`a[m:n:p]`	Selecciona los elementos entre los índices `m` y `n-1` con salto `p`, que puede ser negativo.
`a[::-1]`	Selecciona todos los elementos en orden inverso.

```
Out[5]: 0

In [6]: a[-2]
Out[6]: 8

In [7]: a[3:7]
Out[7]: array([3, 4, 5, 6])

In [8]: a[0:7:2]
Out[8]: array([0, 2, 4, 6])

In [9]: a[::-1]
Out[9]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

In [10]: a[:5]
Out[10]: array([0, 1, 2, 3, 4])

In [11]: a[8:0:-2]
Out[11]: array([8, 6, 4, 2])
```

1.5.4 Direccionamiento de los elementos de un array multidimensional

Con los arrays de dimensión mayor que 1, los elementos se seleccionan como se ha realizado en el apartado anterior, aplicando la sintaxis previa a cada eje (dimensión) del array. Así, si `d` es un array 2-D, la selección del elemento d_{ij} se realizaría en la forma `d[i][j]` o, de manera equivalente (y más común) `d[i,j]`. O bien, `d[2,3] = 6`, que asignaría el 6 al elemento de la fila 2, columna 3 del array `d`. Recuérdese que

en Python se numeran filas y columnas comenzando en 0. Donde se numeran filas y columnas comenzando en 0.

En el caso de una matriz (array 2-D), por ejemplo, se podrá elegir el elemento `j` de la última fila con `d[-1,j]`, el elemento `i` de la última columna con `d[i,-1]` y el último elemento de la matriz con `d[-1,-1]`. Por otro lado, el operador dos puntos (`:`) nos permite seleccionar elementos o bloques de elementos de un array. Es recomendable ejecutar los comandos que se muestran en el siguiente ejemplo para familiarizarse con esta notación que a veces resulta compleja de entender.

Ejemplo 1.5.1

```
In [1]: import numpy as np

In [2]: np.random.seed(8) # Permite cambiar la semilla del generador aleatorio

In [3]: A = np.random.randint(1,50, size=[6,6]) # Generación de una matriz de
     enteros aleatorios siguiendo una distribución uniforme entre [1,50), (no
     incluye el 50). De tamaño 6x6

In [4]: A
Out[4]:
array([[ 4, 21, 42, 6, 27, 9],
       [20, 41, 49, 46, 22, 11],
       [28, 45, 14, 10, 16, 30],
       [35, 19, 15, 39, 30, 10],
       [ 5, 8, 43, 42, 27, 3],
       [39, 10, 38, 47, 18, 44]])

In [5]: A[2,3]
Out[5]: 10

In [6]: A[1,:]  # Selecciona completa la fila 1 (recuérdese que la numeración
     comienza con el 0)
Out[6]: array([20, 41, 49, 46, 22, 11])

In [7]: A[1,]   # Selecciona completa la fila 1 (una expresión equivalente)
Out[7]: array([20, 41, 49, 46, 22, 11])

In [8]: A[:,2]  # Selecciona completa la columna 2
Out[8]: array([42, 49, 14, 15, 43, 38])

In [9]: A[3:5] # Selecciona las filas 3 hasta la 5 (excluyendo la 5). A[3:5,] es
     una expresión equivalente
Out[9]:
array([[35, 19, 15, 39, 30, 10],
       [ 5, 8, 43, 42, 27, 3]])

In [10]: A[:,3:5] # Selecciona las columnas 3 hasta la 5 (excluyendo la 5)
Out[10]:
array([[ 6, 27],
       [46, 22],
       [10, 16],
```

```
       [39, 30],
       [42, 27],
       [47, 18]])

In [11]: A[[1,2,4],:]   # Selecciona las filas 1, 2, 4 completas
Out[11]:
array([[20, 41, 49, 46, 22, 11],
       [28, 45, 14, 10, 16, 30],
       [ 5, 8, 43, 42, 27, 3]])

In [12]: A[5,1:4]   # De la fila 5, selecciona los elementos entre las columnas 1
     a 4, excluyendo el de la columna 4
Out[12]: array([10, 38, 47])

In [13]: A[1:3,3:]  # De las filas 1 a 3 (excluyendo la 3), selecciona los
     elementos de la columna 3 hasta el final
Out[13]:
array([[46, 22, 11],
       [10, 16, 30]])

In [14]: A[-1,:]
Out[13]: array([39, 10, 38, 47, 18, 44])
```

En todos los códigos anteriores se han extraído filas y no columnas de una matriz dada. Se deja como ejercicio modificar el ejemplo mostrado para lograr la extracción de columnas.

El operador dos puntos, `:`, puede utilizarse a ambos lados del operador `=` lo que permite sustituciones de filas o columnas de una matriz por filas y columnas de otra matriz. Obsérvese el siguiente ejemplo:

Ejemplo 1.5.2

```
In [1]: import numpy as np

In [2]: np.random.seed(8)

In [3]:A = np.random.randint(1,50,size=[6,6])

In [4]: B = np.eye(np.shape(A)[0]) #np.shape(A)[0] devuelve el tamaño de la
     dimensión 0 de la matriz A. Recordad que np.shape(A) o A.shape devuelve un
     secuencia (tuple) con el número de elementos de A en cada eje

In [4]: B
Out[4]:
array([[ 1., 0., 0., 0., 0., 0.],
       [ 0., 1., 0., 0., 0., 0.],
       [ 0., 0., 1., 0., 0., 0.],
       [ 0., 0., 0., 1., 0., 0.],
       [ 0., 0., 0., 0., 1., 0.],
       [ 0., 0., 0., 0., 0., 1.]])
```

```
In [5]: B[[1,3,4],:]=A[0:3,:]

In [6]: B
Out[6]:
array([[ 1.,  0.,  0.,  0.,  0.,  0.],
       [ 4., 21., 42.,  6., 27.,  9.],
       [ 0.,  0.,  1.,  0.,  0.,  0.],
       [20., 41., 49., 46., 22., 11.],
       [28., 45., 14., 10., 16., 30.],
       [ 0.,  0.,  0.,  0.,  0.,  1.]])
```

1.5.5 Incremento y reducción de dimensiones de arrays

Existe un conjunto de funciones definidas en la librería `NumPy` que permiten manipular el número de dimensiones o la forma (`shape`) de un array. Por ejemplo, para aumentar el número de dimensiones (ejes) de un array se hace uso de dos rutinas específicas, `np.expand_dims(a, axis)`, que incrementa en el eje deseado el array que recibe como argumento y `np.newaxis`, que aumenta en una la dimensión allí donde se incluya. Esta última se vio en la Subsección 1.4.1 y es equivalente a utilizar `None`. Finalmente, el método `reshape` de la clase `ndarray` permite incluir fácilmente dimensiones. Un poco más adelante se verá su uso para reordenar elementos en diferentes dimensiones. Aquí se ilustra su utilidad para incluir dimensiones. El método `reshape` toma como argumento un conjunto de valores separados por coma y entre paréntesis. Estos valores indican el número de entradas en cada dimensión. Un valor −1 indica que se desconoce cuántas entradas tendrá esa dimensión, calculándose al conocer el número de entradas del resto de dimensiones y descontarlas al número de elementos total. A continuación se incluyen ejemplos de utilización de estas funciones y métodos para añadir dimensiones.

```
In [1]: import numpy as np

In [2]: # Aumenta las dimensiones de array de un elemento

In [3]: A = np.array(5); A.shape
Out[3]: ()

In [4]: R1 = np.expand_dims(A,axis=0); R1.shape
Out[4]: (1,)

In [5]: R2 = A[np.newaxis]; R2.shape
Out[5]: (1,)

In [6]: R3 = A[None]; R3.shape
Out[6]: (1,)

In [7]: R4 = A.reshape(-1,); R4.shape
Out[7]: (1,)

In [8]: # Generamos la segunda dimension, tiene que estar creada la primera
```

```
In [9]: C1 = np.expand_dims(R1,axis=1); C1.shape
Out[10]: (1, 1)

In [10]: C2 = R2[np.newaxis]; C2.shape
Out[10]: (1, 1)

In [11]: C3 = R3[None]; C3.shape
Out[11]: (1, 1)

In [12]: C4 = R4.reshape(1,1); C4.shape
Out[12]: (1, 1)

In [13]: # Aumentamos en una dimensión un array de varios elementos

In [14]: B = np.array([1, 2, 3]); B.shape
Out[14]: (3,)

In [15]: R1 = np.expand_dims(B,axis=0); R1.shape
Out[15]: (1, 3)

In [16]: #Es interesante ver que al forzar incluir la primera dimensión, pasa los
     datos a la segunda

In [17]: R2 = B[np.newaxis]; R2.shape
Out[17]: (1, 3)

In [18]: R3 = B[None]; R3.shape
Out[18]: (1, 3)

In [19]: R4 = B.reshape(-1,); R4.shape
Out[19]: (3,)

In [20]: C1 = np.expand_dims(B,axis=1); C1.shape
Out[20]: (3, 1)

In [21]: C2 = B[:,np.newaxis]; C2.shape
Out[21]: (3, 1)

In [22]: C3 = B[:,None]; C3.shape
Out[22]: (3, 1)

In [23]: C4 = B.reshape(-1,1); C4.shape
Out[23]: (3, 1)

In [24]: # Aumenta en varias dimensiones un array de varios elementos

In [25]: C = np.array([1, 2, 3, 4, 5]); C.shape
Out[25]: (5,)

In [26]: V1 = C[np.newaxis,:,np.newaxis,np.newaxis,np.newaxis]; V1.shape
Out[26]: (1, 5, 1, 1, 1)

In [27]: V2 = C[None,:,None,None,None]; V2.shape
```

```
Out[27]: (1, 5, 1, 1, 1)

In [28]: V3 = C.reshape(1,-1,1,1,1); V3.shape
Out[28]: (1, 5, 1, 1, 1)
```

Para eliminar dimensiones se utiliza `np.squeeze(a[, axis])`, que suprime una o todas las dimensiones del array que sean iguales a uno. De aquí en adelante, los corchetes `[]` señalan la utilización opcional de algún argumento. En el siguiente código se observa cómo operan estas rutinas sobre diferentes arrays. Es importante recordar que el atributo `shape` es una secuencia con tantos términos como ejes tiene el array y que cada uno de los términos de la misma es igual al número de elementos que el array tiene en esa dirección o eje.

```
In [1]: import numpy as np

In [2]: x = np.array([[[0], [1], [2]]]); x.shape
Out[2]: (1, 3, 1)

In [3]: x1 = np.squeeze(x)

In [4]: x1
Out[4]: array([0, 1, 2])

In [5]: x1.shape
Out[5]: (3,)

In [6]: x2 = np.squeeze(x, axis=(0,))

In [7]: x2
Out[7]:
array([[0],
       [1],
       [2]])

In [8]: x2.shape
Out[8]: (3, 1)

In [9]: x3 = np.squeeze(x, axis=(2,))

In [10]: x3
Out[10]: array([[0, 1, 2]])

In [11]: x3.shape
Out[11]: (1, 3)

In [12]: x4 = x.reshape(-1)

In [13]: x4
Out[13]: array([0, 1, 2])

In [14]: x4.shape
Out[14]: (3,)
```

Puede concluirse que aunque hay varios métodos y funciones para expandir y reducir dimensiones en arrays, el método `reshape` sirve para ambas tareas, siendo su uso bastante intuitivo.

1.5.6 Manipulación de dimensiones y formas (`shape`) de arrays

En la Tabla 1.10 se recogen las principales funciones de la librería `NumPy` y métodos[12] propios de la clase `ndarray` que realizan operaciones de manipulación de dimensiones y formas de arrays, entre las que es importante destacar `np.reshape()`, de la que se muestran algunos ejemplos a continuación. A estas funciones habría que añadir las vistas anteriormente en la Tabla 1.8. Resulta conveniente consultar la ayuda del intérprete `IPython` para una descripción más detallada.

Tabla 1.10: Funciones y métodos de `NumPy` para operar sobre dimensiones y formas de arrays.

Función	¿Qué hace?
`np.expand_dims(a, axis)`, `np.newaxis`	Añade un nuevo eje al array `a`.
`np.squeeze(a[, axis])`	Elimina una o todas las dimensiones del array `a` que sean iguales a uno.
`np.ndarray.flatten(a)`, `a.flatten()`	Devuelve una copia del array `a` en una dimensión.
`np.insert(a, indice, valores[, axis])`	Inserta los `valores` en el array `a` de acuerdo con el `indice`.
`np.reshape(a, newshape[,order])`, `a.reshape(, newshape[,order])`	Devuelve un array con los elementos de `a`, con una nueva forma.

```
In [1]: import numpy as np

In [2]: A = np.random.randint(1,50, size = [4,4])

In [3]: A
Out[3]:
array([[15, 47, 35, 37],
       [26, 10, 36, 40],
       [34, 33, 19, 3],
       [15, 42, 11, 34]])
```

[12] En ocasiones resulta difícil distinguir entre una función de la librería y un método de la clase y tal distinción casi tiene interés únicamente desde un punto de vista formal. Por ejemplo, `np.reshape` es una función y `reshape` es un método de la clase `ndarray`.

```
In [4]: A.flatten()
Out[4]: array([15, 47, 35, 37, 26, 10, 36, 40, 34, 33, 19, 3, 15, 42, 11, 34])

In [5]: np.append(A,[[0,0,0],[1,2,3]])
Out[5]:
array([15, 47, 35, 37, 26, 10, 36, 40, 34, 33, 19, 3, 15, 42, 11, 34, 0,
       0, 0, 1, 2, 3])

In [6]: np.append(A, [[0,0,0,0]], axis=0)
Out[6]:
array([[15, 47, 35, 37],
       [26, 10, 36, 40],
       [34, 33, 19, 3],
       [15, 42, 11, 34],
       [ 0, 0, 0, 0]])

In [7]: np.append(A, [[0],[0],[0],[0]], axis=1)
Out[7]:
array([[15, 47, 35, 37, 0],
       [26, 10, 36, 40, 0],
       [34, 33, 19, 3, 0],
       [15, 42, 11, 34, 0]])

In [8]: A.reshape([2,8])
Out[8]:
array([[15, 47, 35, 37, 26, 10, 36, 40],
       [34, 33, 19, 3, 15, 42, 11, 34]])

In [9]: np.reshape(A,[2,8], order='C')
Out[9]:
array([[15, 47, 35, 37, 26, 10, 36, 40],
       [34, 33, 19, 3, 15, 42, 11, 34]])

In [10]: np.reshape(A,[2,8], order='F')
Out[10]:
array([[15, 34, 47, 33, 35, 19, 37, 3],
       [26, 15, 10, 42, 36, 11, 40, 34]])

In [11]: np.arange(6).reshape([3,2])
Out[11]:
array([[0, 1],
       [2, 3],
       [4, 5]])

In [12]: np.arange(6).reshape([3,2], order='F')
Out[12]:
array([[0, 3],
       [1, 4],
       [2, 5]])
```

1.6 Operaciones con arrays

Suponiendo que `A` y `B` sean dos arrays de dimensiones adecuadas, las operaciones más frecuentes con los mismos, junto con su expresión y significado, se muestran en la Tabla 1.11. En el siguiente bloque de código pueden verse algunos ejemplos de estas operaciones.

Tabla 1.11: Operaciones básicas con arrays.

Expresión	Operación
`A + B`, `np.add`	Suma de arrays
`A-B`, np.subtract	Resta de arrays
`A*B`, np.multiply	Producto elemento a elemento de dos arrays
`A/B`, np.divide	División elemento a elemento de dos arrays
`A**B`	Potenciación elemento a elemento de matrices
`A@B`, `np.dot(A,B)`	Producto ordinario de arrays
`np.vdot(a,b)`	Producto interno de dos vectores 1-D. Si `a, b` no son monodimensionales los convierte
`np.linalg.inv(A)`	Inversa del array `A`
`np.linalg.solve(A,B)`	División elemento a elemento de arrays por la izquierda
`np.linalg.matrix_power(A,n)`	Potencia `n`-ésima del array `A`
`A.T`	Transpuesta sin conjugar del array `A`
`A.conj().T`	Transpuesta conjugada del array `A`

```
In [1]: import numpy as np

In [2]: A = np.array([[1,3],[4,2]])

In [3]: A
Out[3]:
array([[1, 3],
       [4, 2]])

In [4]: B = np.array([[1,1],[1,0]])

In [5]: B
Out[5]:
array([[1, 1],
       [1, 0]])
```

```
In [6]: A+B
Out[6]:
array([[2, 4],
       [5, 2]])

In [7]: A*B
Out[7]:
array([[1, 3],
       [4, 0]])

In [8]: E = np.dot(A,B)

In [9]: F = np.dot(A,np.linalg.inv(B))

In [10]: F
Out[10]:
array([[ 3., -2.],
       [ 2., 2.]])

In [11]: np.dot(F,B)
Out[11]:
array([[1., 3.],
       [4., 2.]])

In [12]: G = np.linalg.solve(A,B)

In [13]: np.dot(A,G)
Out[13]:
array([[1., 1.],
       [1., 0.]])

In [14]: H=A+1j*B

In [15]: H
Out[15]:
array([[1.+1.j, 3.+1.j],
       [4.+1.j, 2.+0.j]])

In [16]: H.conjugate().T
Out[16]:
array([[1.-1.j, 4.-1.j],
       [3.-1.j, 2.-0.j]])

In [17]: H.T
Out[17]:
array([[1.+1.j, 4.+1.j],
       [3.+1.j, 2.+0.j]])

In [18]: np.linalg.matrix_power(A,2)
Out[18]:
array([[13, 9],
       [12, 16]])
```

```
In [19]: A**2
Out[19]:
array([[1, 9],
       [16, 4]])

In [20]: A**B # Potenciación elemento a elemento de A: M[i,j]=A[i,j]^B[i,j]
Out[20]:
array([[1, 3],
       [4, 1]])

In [21]: a = np.array([1,2,3])

In [22]: a
Out[22]: array([1, 2, 3])

In [23]: np.vdot(a,a)
Out[23]: 14
```

Es interesante observar que la sentencia `np.linalg.matrix_power(A,2)` podría sustituirse por su equivalente, `A**2`. Uno de los ejemplos anteriores ha consistido en la operación de división por la izquierda, que se utiliza (entre otras cosas) para la solución de un sistema de ecuaciones lineales, como puede verse más claramente en lo que sigue. Se desea resolver el sistema de ecuaciones dado por:

$$\begin{aligned} x + 2y - 3z &= 5 \\ -3x - y + z &= -8 \\ x - y + z &= 0 \end{aligned}$$

que puede escribirse en forma matricial como $\mathbf{Ax} = \mathbf{b}$, con las matrices $\mathbf{A}$,$\mathbf{x}$,$\mathbf{b}$ dadas por:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & -3 \\ -3 & -1 & 1 \\ 1 & -1 & 1 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 5 \\ 8 \\ 0 \end{bmatrix}$$

La escritura y solución del sistema de ecuaciones anterior se muestra en el siguiente código:

```
In [1]: import numpy as np

In [2]: A = np.array([[1,2,-1],[-3,-1,1],[1,-1,1]])

In [3]: A
Out[3]:
array([[ 1, 2, -1],
       [-3, -1, 1],
       [ 1, -1, 1]])

In [4]: b = np.array([[5],[8],[0]])
```

```
In [5]: b
Out[5]:
array([[5],
       [8],
       [0]])

In [6]: x = np.linalg.solve(A,b)

In [7]: x
Out[7]:
array([[ -2.],
       [  9.],
       [ 11.]])
```

1.6.1 Texto

Una cadena de caracteres es cualquier texto rodeado por comillas simples o dobles. Si se desea que algo se escriba en la pantalla, se puede utilizar el comando `print`. Para imprimir texto junto al valor de la variable se puede utilizar un formato similar al utilizado en el lenguaje de programación C (con `printf`) o Matlab (con `sprintf`), utilizando el % seguido de números para indicar las cifras significativas y una letra que identifica el tipo de número. Como por ejemplo:

```
a = 3.4235
b = 7
print('A vale %.2f y b vale %d' % (a,b)) #f float, d decimal
```

De forma equivalente, en Python el comando `.format()` permite escribir un texto formateado por pantalla o en un fichero de texto. Para entender su uso se pueden ejecutar las siguientes líneas de código:

```
a = 3.4235
b = 7
print('A vale {:.2f} y b vale {}'.format(a,b))
```

1.7 Operadores relacionales y lógicos

El lenguaje Python dispone del conjunto de operadores relacionales que se muestran en la Tabla 1.12.

Si el resultado de una comparación se cumple, el resultado es verdadero y se representa por `1` y `0` en caso contrario. Cuando la comparación se realiza entre arrays de dimensiones adecuadas (puesto que la comparación se hace entrada a entrada los dos arrays deben tener las mismas dimensiones), el resultado será otro array booleano, como puede verse en el siguiente ejemplo:

Tabla 1.12: Operadores relacionales.

Operador	Significado
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
==	Igual a
! =	Distinto de

```
In [1]: import numpy as np

In [2]: A = np.array([[1,2],[3,4]]); B=np.array([[1,0],[5,4]])

In [3]: A
Out[3]:
array([[1, 2],
       [3, 4]])

In [4]: B
Out[4]:
array([[1, 0],
       [5, 4]])

In [5]: A==B
Out[5]:
array([[ True, False],
      [False, True]])

In [6]: A>B
Out[6]:
array([[False, True],
      [False, False]])

In [7]: A!=B
Out[7]:
array([[False, True],
      [ True, False]])

In [8]: np.sum(A!=B)  # Cuenta cuántos elementos son distintos
Out[8]: 2
```

Con la última instrucción, `np.sum(A!=B)`, se está calculando el número de elementos distintos entre los dos arrays. Este código se utilizará en capítulos posteriores, por ejemplo, para calcular el número de errores en una transmisión digital.

Los operadores lógicos de NumPy se muestran en la Tabla 1.13. Y en el siguiente bloque de código, algunos ejemplos de estos operadores.

Tabla 1.13: Operadores lógicos.

Operador	Significado
`np.logical_and(a,b)`	Función lógica *and* punto a punto entre arrays `a` y `b`
`np.logical_or(a,b)`	Función lógica *or* punto a punto entre arrays `a` y `b`
`np.logical_not(a)`	Negación lógica punto a punto

```
In [1]: import numpy as np

In 2: a = np.arange(10); b = 7-a

In [3]: a
Out[3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [4]: b
Out[4]: array([ 7, 6, 5, 4, 3, 2, 1, 0, -1, -2])

In [5]: r1 = np.logical_and(a>b, b>2)

In [6]: r1
Out[6]: array([False, False, False, False, True, False, False, False, False,
     False])

In [7]: r2 = np.logical_or(a<b, b<2)

In [8]: r2
Out[8]: array([ True, True, True, True, False, False, True, True, True, True])

In [9]: r3 = np.logical_not(r2)

In [10]: r3
Out[10]: array([False, False, False, False, True, True, False, False, False,
     False])
```

Utilización de la función `np.where`

Por su interés, a continuación se muestra la utilización de una de las funciones lógicas más versátiles de la librería NumPy, la función `np.where(x)`.

Dado un vector `x`, la función `np.where(x)` devuelve los índices en los que el vector `x` es distinto de cero. En las siguientes líneas se utiliza de manera elemental.

```
In [1]: import numpy as np

In [2]: x = np.array([[3, -1, 4, 0, -8, 0, 2]])

In [3]: np.where(x)
```

```
Out[3]: (array([0, 0, 0, 0, 0]), array([0, 1, 2, 4, 6]))
```

Pero, además, la función `np.where(x)` permitiría a su vez definir un array formado únicamente con los elementos de `x` que son distintos de cero:

```
In [4]: y=x[np.where(x)]

In [5]: y
Out[5]: array([[ 3, -1, 4, -8, 2]])
```

o bien encontrar las posiciones de los elementos de un array que cumplen una determinada condición, como ser mayores que un determinado valor; y a su vez un array formado por esos elementos, como el array `z` del siguiente código:

```
In [1]: import numpy as np

In [2]: x = np.array([[3, -1, 4, 0, -8, 0, 2]])

In [3]: np.where(x>=2)
Out[3]: (array([0, 0, 0]), array([0, 2, 6]))

In [4]: z=x[np.where(x>=2)]

In [5]: z
Out[5]: array([[3, 4, 2]])
```

1.8 Gráficos

La visualización de gráficos en Python se realiza mediante los recursos de la librería `Matplotlib` construida sobre arrays de `Numpy`; esto es, se realiza una representación gráfica en dos y tres dimensiones aunque en realidad lo que se hace es representar unos arrays frente a otros, al igual que ocurría en MATLAB. Una descripción en profundidad de la librería `Matplotlib` y sus funciones puede encontrarse en los textos ya citados [Joh19] y [Van16b], que se utilizarán en este apartado. Referencias más específicas pueden encontrarse en los textos [Dev14], [Tos09], [AY18] y [McG15]. Una galería con múltiples ejemplos se encuentra en https://matplotlib.org/gallery/index.html.

1.8.1 Aspectos generales de `Matplotlib`

El primer paso para el uso de la librería `Matplotlib` consistirá en su instalación e importación, mediante el procedimiento descrito en el apartado 1.3.5. Por ello, las primeras líneas de código habituales para su uso consistirán en la importación de la librería y algunos de sus módulos, utilizando los alias `mpl` y `plt`, según el uso habitual.

`Matplotlib` se escribió originalmente como una alternativa a MATLAB, por lo que mucho de su sintaxis recuerda este hecho. Esto hace que los usuarios de MATLAB puedan construir con facilidad y rapidez diagramas simples sin demasiada dificultad. Esta

interfase, que será la utilizada principalmente en este texto, es rápida y conveniente para gráficos simples pero no es la más adecuada en situaciones más complejas.

Un procedimiento más eficiente consiste en explotar adecuadamente la programación orientada a objeto (*object oriented programming*, OOP) propia de Python, haciendo uso explícito de los *métodos* asociados a los objetos de las clases `Figure` y `Axes` introducidos por la librería `Matplotlib`[13]. Más adelante se ofrecerán ejemplos de los dos procedimientos mencionados aunque, como ocurría en el caso de la librería `NumPy`, muchas de las funciones existentes tienen su contrapartida en métodos asociados a las clases definidas en la librería `Matplotlib`. Dependiendo del entorno de programación que se esté utilizando, podría se necesario hacer uso de una importante función de `Matplotlib`, responsable de mostrar el gráfico construido: la función `plt.show()`.

Los siguientes códigos usando `IPython` darían el resultado mostrado en la Figura 1.2. Nótese que ambos son equivalentes, uno utilizando las funciones, y por lo tanto, la función `plt.show()`, y el otro usando los métodos de las clases. La primera de las sentencias, `%matplotlib`, puede no ser necesaria dependiendo de la versión de la librería instalada.

```
In [1]: %matplotlib
   ...: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...:
   ...: t = np.arange(0,5*np.pi,0.1)
   ...: plt.figure()
   ...: plt.plot(t,np.sin(t))
   ...: plt.show()
```

```
In [2]: %matplotlib
   ...: import matplotlib.pyplot as plt
   ...: import numpy as np
   ...:
   ...: fig = plt.figure()
   ...: ax = plt.axes()
   ...: t = np.arange(0,5*np.pi,0.1)
   ...: ax.plot(t,np.sin(t))
```

En este último caso se ha utilizado el segundo de los procedimientos de uso de la librería `Matplolib`, mediante una OOP, creando las variable `fig`, igual a una instancia de la clase `plt.Figure`, y la variable `ax`, igual a una instancia de la clase `plt.Axes`[14]. Los métodos y funciones asociados a estos objetos permiten la obtención de gráficos con una sintaxis diferente como puede observarse en el mencionado código. Como ejemplo adicional, para salvar la figura generada en el código anterior, la Figura 1.2,

[13] Ver [Joh19, págs. 138-140] para una definición de cada una de estas dos clases.

[14] En realidad, esto no es rigurosamente cierto, como puede comprobarse determinando el tipo de la variable `ax`, pero suministra una idea válida para el uso de la interfaz OOP.

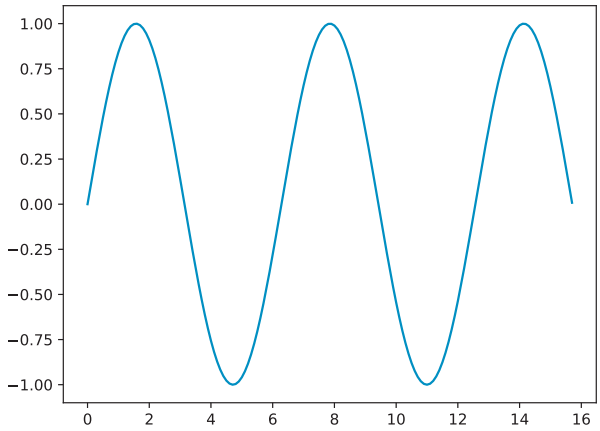

Figura 1.2: Ejemplo de un gráfico creado con `Matplotlib`.

se ha ejecutado la instrucción `fig.savefig('nombre-del-fichero.pdf')`, que utiliza el *método* `savefig` del objeto `fig`. Este método toma una cadena de caracteres con el nombre del fichero como primer argumento. Por defecto, el formato del fichero de salida se determina mediante la extensión del nombre del fichero, aunque también puede definirse explícitamente con algunos de los argumentos adicionales que admite el método `savefig`.

Para acabar este apartado introductorio de la librería `Matplotlib`, mencionar la existencia de *estilos* predefinidos para la realización de gráficos. En las líneas de código que siguen se muestra el uso de uno de estos estilos y en la Figura 1.3 se muestra la correspondiente salida. Una relación de los estilos existentes se recoge en la lista `plt.style.available` y una versión de la misma figura, con el estilo `ggplots` se muestra en la Figura 1.4.

```
In [1]: %matplotlib
Using matplotlib backend: MacOSX

In [2]: import matplotlib.pyplot as plt

In [3]: import numpy as np

In [4]: plt.style.use('seaborn-whitegrid')

In [5]: fig, ax = plt.subplots()

In [6]: t = np.arange(0,5*np.pi,0.1)

In [7]: ax.plot(t,np.sin(t))
Out[7]: [<matplotlib.lines.Line2D at 0x7f9c50455070>]
```

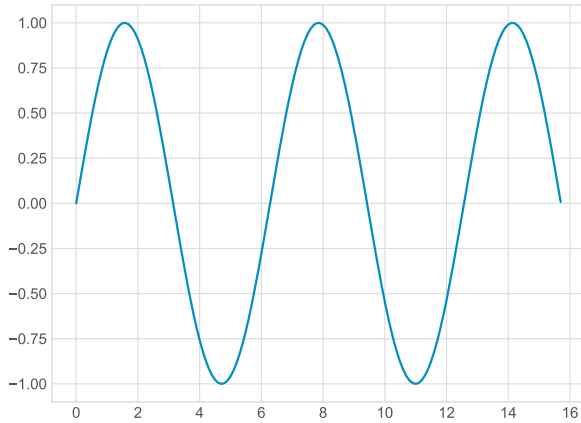

Figura 1.3: Ejemplo de un gráfico creado con `Matplotlib` y estilo `seaborn-whitegrid`.

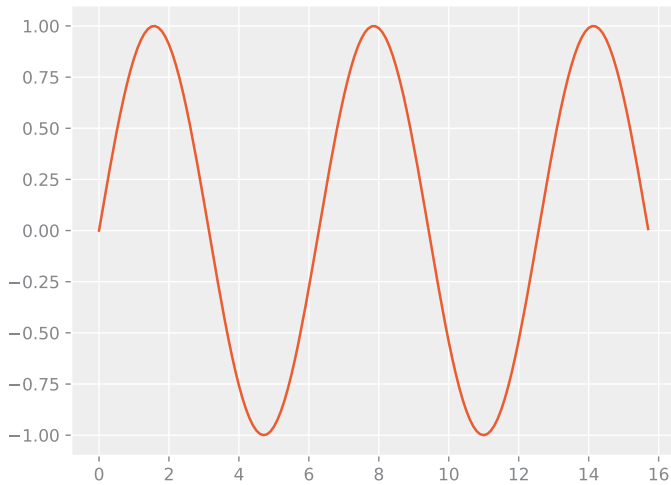

Figura 1.4: Ejemplo de un gráfico creado con `Matplotlib` y estilo `ggplot`.

1.8.2 Funciones gráficas elementales 2D

Las funciones gráficas elementales 2D se muestran en la Tabla 1.14, utilizando un formato de llamada similar al usado en MATLAB. Existen, además, un conjunto de funciones orientadas a colocar etiquetas o manipular los gráficos construidos, algunas de las cuales se muestran en la Tabla 1.15. A continuación se describen algunas de ellas.

Tabla 1.14: Funciones gráficas elementales 2D.

Función	Significado
`plt.stem(N)`	Representa la secuencia discreta `N`.
`plt.plot(x,y)`	Crea un gráfico a partir de los vectores `x` y `y` con escalas lineales donde los dos vectores tienen que tener la misma dimensión.
`plt.semilogx(x,y)`	ídem con escala lineal en el eje `y` y logarítmica en el eje `x`.
`plt.semilogy(x,y)`	ídem con escala lineal en el eje `x` y logarítmica en el eje `y`.

1.8.2.1 Función `stem`

En su expresión más simple, `plt.stem(Y)`, permite representar la secuencia discreta `Y`, mientras que `plt.stem(N,Y)` construye una representación de la secuencia discreta `Y` en los valores dados por `N`. `N` e `Y` tienen que tener la misma dimensión. Por ejemplo, el siguiente código, en el que se ha prescindido de las importaciones de las librerías,

```
N = np.arange(60)
plt.stem(np.sin(N*np.pi/12))
plt.axis([-1,61,-1.1,1.1])
plt.grid()
plt.show()
```

daría como resultado la gráfica mostrada en la Figura 1.5. Se pueden especificar los marcadores, tipos de línea y color para la señal representada, y el color y tipo de línea para ejes (base), de acuerdo con el siguiente código, por ejemplo,

```
N = np.arange(60)
plt.stem(np.sin(N*np.pi/12),linefmt='b-', markerfmt='bo',basefmt='r-',
     use_line_collection = True)
plt.axis([-1,61,-1.1,1.1])
plt.grid()
plt.show()
```

Tabla 1.15: Funciones de manipulación de gráficos.

Función	Significado
`plt.show()`	Muestra la gráfica actual
`plt.xlabel('texto')`	Etiqueta el eje x de la gráfica actual
`plt.ylabel('texto')`	Etiqueta el eje y de la gráfica actual
`plt.title('texto')`	Título de la gráfica actual
`plt.text(x,y, 'texto')`	Introduce "texto" en la posición `(x,y)` de la gráfica actual
`plt.legend()`	Permite definir rótulos para las distintas líneas o curvas de la gráfica
`plt.grid()`	Dibuja una rejilla, con `grid off` desaparece la cuadrícula
`plt.axis([xmin, xmax, ymin, ymax])`	Fija los valores máximos y mínimos de los ejes
`plt.axis('equal')`	Establece que la escala de los ejes sea la misma
`plt.axis('square')`	Fija que la gráfica sea un cuadrado
`plt.hold(True)`	Permite volver a representar sobre la gráfica actual una nueva gráfica
`plt.hold(False)`	Desactiva lo anterior
`plt.close()`	Cierra la figura activa
`plt.close('all')`	Cierra todas las figuras

donde el último argumento evita la aparición de un "warning", dependiendo de la versión.

1.8.2.2 Función `plot`

Es la función clave para la representación de un gráfico de dos dimensiones y también un método asociado a la clase `Axes`. En sus diversas variantes no hace más que representar un vector frente a otro de la misma dimensión, con un conjunto de parámetros adicionales que controlan la forma de la representación, el color, etc., como se ha podido observar en las Figuras 1.2 y 1.3. Analizando cualquiera de las mismas, hay que recordar que lo que realmente se ha hecho es representar los puntos `y=sin(t)` frente a `x=t`. Esos puntos, por defecto, se han unido por una segmento que en este caso ha sido una línea recta azul. Este comportamiento por defecto puede modificarse con un conjunto de parámetros que controlan el color, el estilo de línea y el marcador que se utiliza para señalar el punto `(x,y)`. Así, por ejemplo, las lineas:

```
t = np.arange(0,5*np.pi,0.1)
plt.plot(t,np.sin(t), '*-r')
plt.show()
```

generarían el gráfico mostrado en la Figura 1.6. Si la opción utilizada hubiera sido `'r*'` se habrían representado únicamente los puntos sin unirlos.

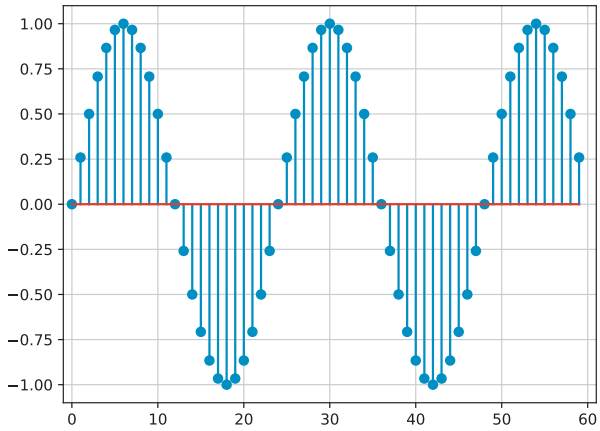

Figura 1.5: Gráfica obtenida en Python con la función `plt.stem` de la librería `Matplotlib`.

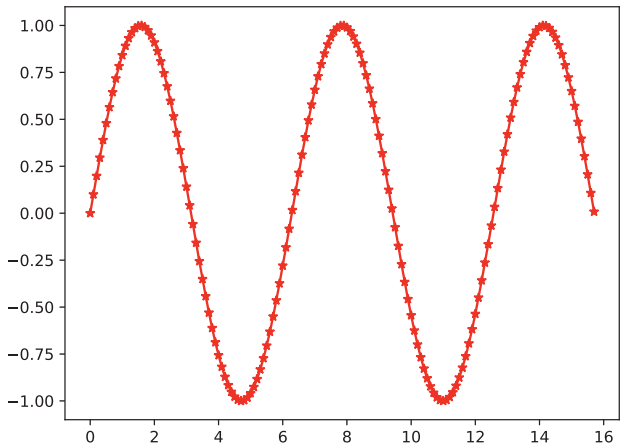

Figura 1.6: Gráfica obtenida en Python con la función `plot` de la librería `Matplotlib`.

Adicionalmente también es posible añadir a la función `plot` algunos especificadores de línea que controlan el espesor de la línea, el tamaño de los marcadores, etc. Por ejemplo, resulta interesante ejecutar el siguiente código:

```
t = np.arange(0,5*np.pi,0.1)
plt.plot(t,np.sin(t),'-.rs', linewidth=4, markeredgecolor='k', markerfacecolor='g
    ', markersize=10)
plt.show()
```

Por defecto Python añade curvas o gráficos sobre la misma figura mientras no se cambie de figura con el comando `plt.figure(n)` siendo `n` el número que se le da a una figura para tener el control de las mismas o se cree una nueva instancia de la clase `Figure`. El siguiente código representa en la Figura 1.7 dos curvas en el mismo plot.

```
t = np.arange(0,5*np.pi,0.1)
plt.plot(t,np.sin(t))
plt.plot(t,np.sin(t)**2-1)
plt.show()
```

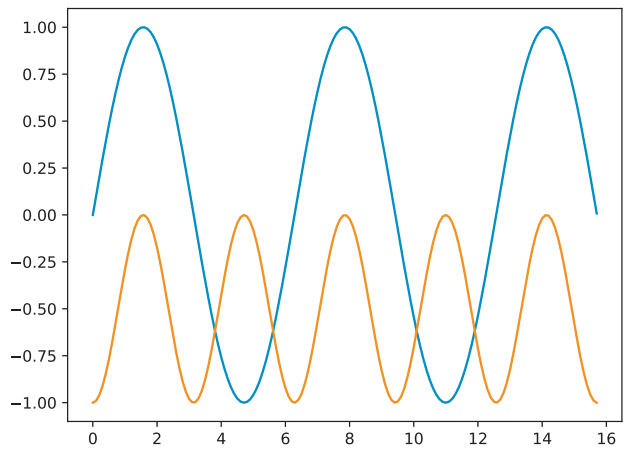

Figura 1.7: Gráfica obtenida en Python con la función `plot` de la librería `Matplotlib`.

Por último, en la Figura 1.8 se muestra el resultado del uso de algunos de los comandos que permiten etiquetar los ejes, dar un título al gráfico, definir rótulos para las distintas curvas que se representan y dibujar una cuadrícula en el mismo. Nuevamente no se ha mostrado la importación de las librerías usadas.

```
x = np.linspace(0,10,1000)
y1 = (x/2)**2
y2 = x**2
y3 = 4 + (x/2)**2
```

```
plt.plot(x,y1,x,y2,x,y3)
plt.title('Polinomios de x')
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.legend(['$(x/2)^2$','$x^2$','$4 + (x/2)^2$'])
plt.grid()
plt.show()
```

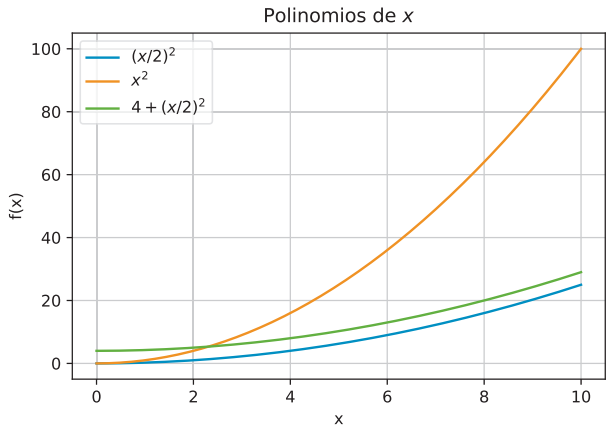

Figura 1.8: Otro ejemplo de la función `plot` de la librería `Matplotlib` .

Una alternativa para la generación de la Figura 1.8 mediante la interfase OOP se muestra a continuación (aquí se ha mostrado la importación de las librerías).

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = np.linspace(0,10,1000)
y1 = (x/2)**2
y2 = x**2
y3 = 4 + (x/2)**2
ax.plot(x,y1,x,y2,x,y3)
ax.set(xlabel='x', ylabel='f(x)', title='Polinomios de $x$')
ax.legend(['$(x/2)^2$','$x^2$','$4 + (x/2)^2$'])
ax.grid()
```

1.8.2.3 Comando `subplot`

Mediante el comando `subplot`, una ventana gráfica se puede dividir en (`m` × `n`) particiones con el objetivo de representar múltiples gráficos en las mismas. Cada una de estas subventanas tiene sus propios ejes aunque otras propiedades son comunes a toda la figura. La forma general de este comando es `subplot(m,n,k)` donde `m` y `n` señalan el número de subdivisiones y `k` es el número de la subdivisión, empezando la numeración por la primera fila, de izquierda a derecha, después la segunda, etc. El

siguiente código y la salida correspondiente, mostrada en la Figura 1.9 muestran su funcionamiento.

```
t = np.arange(0,5*np.pi,0.1)
plt.subplot(2,2,1), plt.title('$Sen(x)$'), plt.plot(t,np.sin(t))
plt.subplot(2,2,2), plt.title('$Cos(x)$'), plt.plot(t,np.cos(t))
plt.subplot(2,2,3), plt.title('$Sen^{2}(x)$'), plt.plot(t,np.sin(t)**2)
plt.subplot(2,2,4), plt.title('$Cos^{2}(x)$'), plt.plot(t,np.cos(t)**2)
plt.tight_layout() # Para que no se solapen las subfiguras
#Alternativamente se puede usar:
#plt.subplots_adjust(hspace=0.5)
plt.show()
```

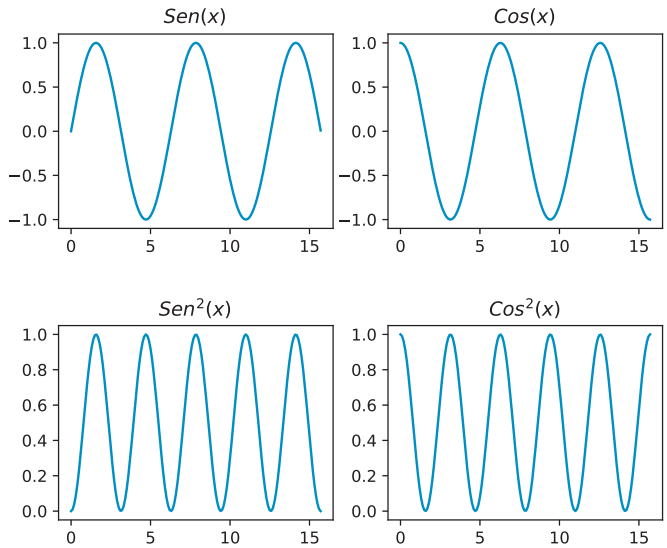

Figura 1.9: Gráfica obtenida en Python con la función `subplot` de la librería `Matplotlib`.

1.8.3 Funciones gráficas elementales 3D

El submódulo `Axes3D` incluido dentro de las herramientas `mpl_toolkits.mplot3d` de la librería `Matplotlib` aporta los métodos necesarios para crear diagramas 3D dentro de Python. En las líneas de código siguientes se realiza la importación del submódulo y sobre la instancia `fig` se representan dos instancias de objetos `Axes`: una de dos dimensiones y otra de tres, haciendo uso del método `add_subplot()`. En esta ocasión la interfase de programación ha sido OOP.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fi = np.arange(0,6*np.pi,np.pi/20)
fig = plt.figure()
ax2d = fig.add_subplot(1, 2, 1)
ax3d = fig.add_subplot(1, 2, 2, projection='3d')

ax2d.plot(np.cos(fi),np.sin(fi)), ax2d.grid(), ax2d.axis('square')
ax3d.plot(np.cos(fi),np.sin(fi),fi), ax3d.grid()
```

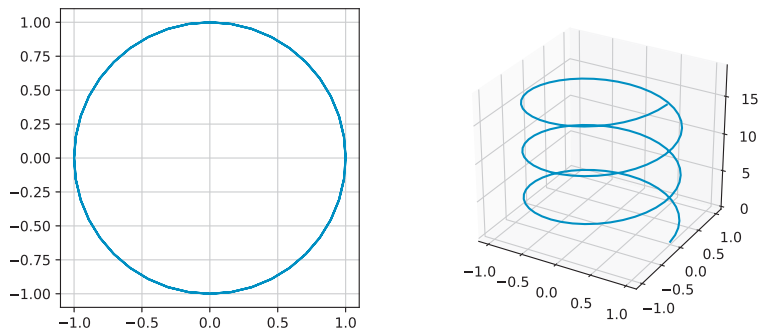

Figura 1.10: Gráfica 3D obtenida en Python con la función `subplot` de la librería `Matplotlib`.

En la representación 3D, el comando más utilizado es `plot_surface(X, Y, Z)`, en el que se representa de manera tridimensional `Z` frente a `(X, Y)`. Para una adecuada representación, en primer lugar se necesita construir una rejilla de valores del par `(X, Y)`, lo que se logra mediante el uso del comando `meshgrid(x,y)`, siendo `x` e `y` dos arrays que representan los valores en los ejes para los que se desea construir la rejilla. En el siguiente ejemplo, bastante autoexplicativo, se representa la función densidad de probabilidad gaussiana de dos dimensiones, con media cero y varianza unidad.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.gca(projection='3d')

# Crea la rejilla de valores X, Y cuando x varía entre (-3,3) e y varía entre (-3,
    3)
x = np.arange(-3, 3, 0.01)
y = np.arange(-3, 3, 0.01)
```

```
X, Y = np.meshgrid(x, y)

# Definición de la función a representar
Z=(1/(2*np.pi))*np.exp(-X**2-Y**2)*0.5

ax.plot_surface(X, Y, Z, cmap=plt.get_cmap('Spectral_r'))
```

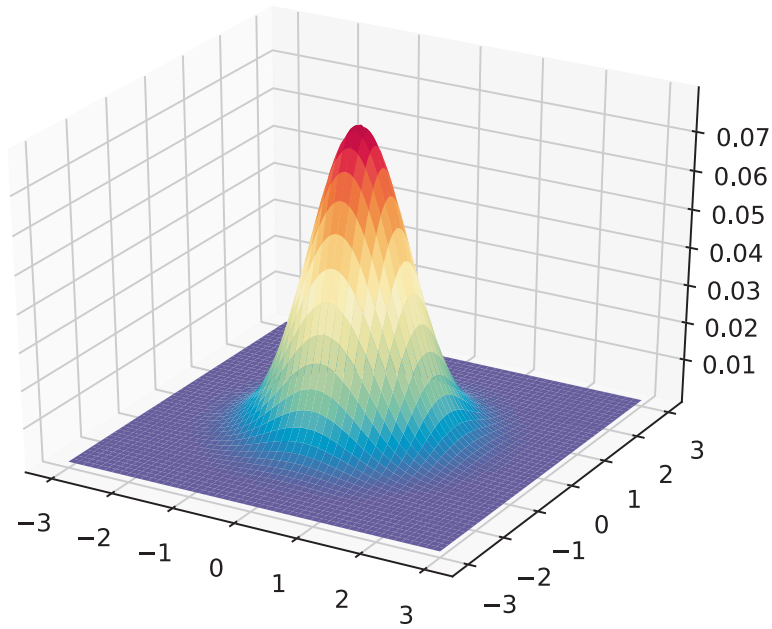

Figura 1.11: Gráfica 3D obtenida en Python con la función `plot_surface` de la librería `Matplotlib`.

Otro ejemplo, el logotipo de Octave, se obtiene con el siguiente código. El resultado de su ejecución se muestra en la Figura 1.12.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.gca(projection='3d')

# Crea la rejilla de valores X, Y cuando x varía entre (--10,10) e y varía entre
    (-10,10)
x = np.arange(-10, 10, 0.03)
y = np.arange(-10, 10, 0.03)
```

```
X, Y = np.meshgrid(x, y)

R=np.sqrt(X**2 + Y**2)
Z=np.sin(R)/R
ax.plot_surface(X, Y, Z)
ax.plot_surface(X, Y, Z, cmap=cm.Spectral_r)
```

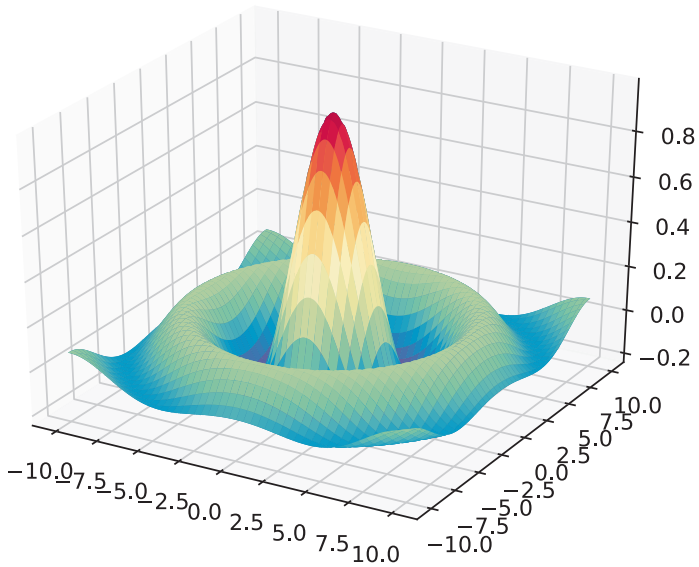

Figura 1.12: Logotipo de Octave.

En este apartado se han revisado brevemente las funciones y métodos más habituales de la librería `Matplotlib`. Puede resultar interesante ampliar los conocimientos con el Capítulo 4 del texto *Python Data Science Handbook*.

1.9 Programación en Python: bifurcaciones y bucles

Como cualquier otro lenguaje de programación, Python contiene bifurcaciones que permiten realizar una u otra operación e instrucciones que permiten la realización repetitiva de una determinada acción. El cuerpo de las bifurcaciones y bucles debe estar indentado con respecto a la sentencia de inicio del elemento. Esta indentación es la forma que tiene el intérprete de Python de detectar el cuerpo de estos elementos. En el caso del intérprete `IPython`, esta indentación aparece como una secuencia de tres puntos '. . .'. En el caso de edición de scripts, esta indentación aparece como espacios en blanco.

1.9.1 Sentencia `if`

En su forma más simple, la sentencia `if` se escribe en la forma:

```
if condicion:
  sentencias
```

Aunque existe también la bifurcación múltiple que puede concatenarse como se desee, como se observa en el siguiente código:

```
if condicion1:
  bloque1      # Indentación
elif condicion2:
  bloque2
elif condicion3:
  bloque3
else:          # Opción por defecto cuando no se cumplen las condiciones 1, 2, 3
  bloque4
```

Por ejemplo, al ejecutarse

```
condicion1 = 0
condicion2 = 1
condicion3 = 1 #No se comprueba más abajo si alguna anterior no es cero
bloque1 = 1,
bloque2 = 2,
bloque3 = 3,
bloque4 = 4

if condicion1:
    print(bloque1) # Indentación
elif condicion2: print(bloque2) #linea continua
elif condicion3:
    print(bloque3)
else: # Opción por defecto cuando no se cumplen las condiciones 1, 2, 3
    print(bloque4)
```

se obtendría

```
(2,)
```

1.9.2 Sentencia `while`

La sentencia `while` es la construcción más general de repetición del lenguaje. En su expresión más simple, un conjunto de código se repite mientras que la condición inicial se cumpla, sintaxis que se muestra a continuación.

```
while condicion:    # Condición de prueba
  sentencias        # Cuerpo del bucle
```

También es posible esta construcción optativa:

```
while condicion:    # Condición de prueba
  sentencias        # Cuerpo del bucle
else:               # Una parte opcional
  sentencias        # Se ejecutarán si no se ha salido del bucle con break
```

Un ejemplo sencillo es el siguiente:

```
numero = 0
suma = 0
while numero <= 10:
    suma = numero + suma
    numero = numero + 1
print ("La suma es " + str(suma))
```

1.9.3 Sentencia `for`

La sentencia `for` repite un conjunto de sentencias un determinado número de veces, en función de una determinada condición. En su sintaxis más simple se escribe en la forma:

```
for k in objeto:  # Asigna a k los elementos del objeto
  sentencias  # Cuerpo que se repite. Utiliza k
```

A continuación se propone un ejemplo.

```
objeto = np.arange(5)
for k in objeto: # Asigna a k los elementos del objeto
    print(sentencia) # Cuerpo que se repite. Utiliza k
```

También es posible esta construcción optativa:

```
for k in objeto:      # Asigna a k los elementos del objeto
  sentencias          # Cuerpo que se repite. Utiliza k
else:                 # Una parte opcional
  sentencias          # Se ejecutarán si no se ha salido con break
```

donde `objeto` es cualquier tipo de datos que contenga distintos elementos que va tomando la variable `k`, para cada uno de los cuales se ejecuta el conjunto de sentencias que hay indentadas tras el `for` . Como ejemplo, el siguiente código permite sumar los números pares menores que 100:

```
s = 0
for k in range(2,100,2):
    s = s+k
print(s)
```

En este otro ejemplo se imprimen distintos nombres de modulaciones, donde el cuerpo del bucle se incluye a continuación de los dos puntos:

```
T = ('PAM', 'QAM', 'PSK', 'OQPSK')
for k in T: print(k, end = ' ')
```

En Python en el bucle `for` se puede usar cualquier objeto que sea *iterable*, no sólo aquellos creados mediante la función `range`. Un objeto en Python es *iterable* si posee el atributo `__iter__`. Para conocer si un objeto es iterable o no se puede utilizar la función nativa de Python `iter()` que devolverá la sentencia `'x' object is not iterable` en el caso de que no lo sea. En el siguiente código:

```
A = np.array([[1,2,-1],[-3,-1,1],[1,-1,1]])
for k in A:
  print(k)
[ 1  2 -1]
[-3 -1  1]
[ 1 -1  1]
```

el bucle se ha ejecutado un total de 3 veces, asignando en cada uno de los bucles a `k` un array 1-D correspondiente a cada una de las filas del array 2-D `A`. Para entender porqué se ha ejecutado únicamente un total de 3 veces el bucle y no se ha ido recorriendo elemento a elemento el array 2-D, puede ejecutarse el siguiente código, en el que se crea un nuevo objeto de Python de la clase *iterator*, `it` . Dicho objeto tiene asociado el método `next()`. Su funcionamiento se deduce fácilmente, [Ram15], Capítulo 14.

```
import numpy as np

A = np.array([[1,2,-1],[-3,-1,1],[1,-1,1]])

it = iter(A)

next(it)
Out[4]: array([ 1, 2, -1])

next(it)
Out[5]: array([-3, -1, 1])

next(it)
Out[6]: array([ 1, -1, 1])

next(it)
Traceback (most recent call last):

  File "<ipython-input-7-a883b34d6d8a>", line 1, in <module>
    next(i)

StopIteration
```

Por último, tanto en bucles `for` como en `while` se puede utilizar la sentencia `break` que hace que se termine la ejecución del bucle.

1.10 Ficheros de comandos y funciones

Los ficheros de comandos y funciones son ficheros de texto sin formato que constituyen el núcleo de la programación en Python. En lugar de introducir las órdenes de Python de manera interactiva en el intérprete `IPython`, por ejemplo, recogen un conjunto de órdenes agrupadas en un fichero con extensión `*.py` y se ejecutan desde el directorio de trabajo.

Básicamente existen dos tipos de ficheros `*.py`, los que se pueden denominar *ficheros de comandos* (*scripts* en inglés) y ficheros de funciones. Los primeros constituyen en realidad un conjunto de comandos que se ejecutan sucesivamente cuando se teclea el nombre del fichero en el prompt de Python y los segundos permiten definir funciones y clases equivalentes a las propias de Python. Estas funciones y clases se pueden importar desde otros ficheros utilizando las órdenes `import` en sus diferentes versiones como se ha explicado con anterioridad. A continuación se verá con algo más de detalles cada uno de ellos.

1.10.1 Ficheros de comandos: scripts

Los ficheros de comandos son ficheros con un nombre tal como `fichero01.py` que contienen una sucesión de comandos que se ejecutarán de manera consecutiva como si se escribieran uno tras otro de manera interactiva. Un fichero `*.py` puede llamar a otros ficheros o incluso llamarse a si mismo.

Tal como se introdujo en la Subsección 1.3.1, cuando se desea realizar algún comentario en un fichero de comandos, se utiliza el símbolo `#`, de manera que todo lo que le sigue, hasta la siguiente línea, no es un código ejecutable. Si se desea comentar un conjunto de líneas contiguas se puede optar por empezarlas y terminarlas con una triple comilla.

Las variables definidas en los ficheros de comandos son variables del espacio de trabajo desde el que se ejecuta el fichero; esto es, son variables que tienen el mismo carácter que las creadas de manera interactiva en Python. Al finalizar la ejecución del *script* dichas variables permanecerán en memoria. Esto conlleva la ventaja, para pequeños diseños de códigos, de poder obtener de manera manual el valor de las variables directamente en el workspace, y poder así depurar el código.

Por último, para conocer las variables que se tienen definidas en el espacio de trabajo se utiliza el comando `who`. Si se quieren eliminar alguna de ellas, se ejecuta `del` seguido del nombre de la variable que se desee eliminar, por ejemplo `del variable01` y si se quiere limpiar el espacio de trabajo es necesario reiniciar el kernel, mediante la sentencia `reset`.

1.10.2 Funciones definidas por el usuario

En Python las funciones se definen de la siguiente forma:

```
def mifuncion(entrada01, entrada02, entrada03):
  # cuerpo de la función
  return [salida01, salida02]
```

donde `entrada01`, `entrada02`, `entrada03` son los valores de llamada a la función y `salida01`, `salida02` son los valores que devolverá la función.

A modo de ejemplo se define a continuación una nueva función que se utilizará a lo largo del texto profusamente. En la librería `scipy.special` está disponible la función erfc(x). Sin embargo, en el curso de Comunicaciones Digitales no se acostumbra a utilizar esta función sino la función Q(x) definida en la forma:

$$\mathrm{Q}(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty \mathrm{e}^{-\frac{t^2}{2}}\, \mathrm{d}t = \frac{1}{2}\,\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right),$$

por ello, es habitual definir la función `Qfunct` que devuelve los valores de la función Q(x) mediante un fichero que se denominará, por ejemplo, `Qfunct.py`:

Código 1.2: Definición de la función Q(x).

```
1  def Qfunct(x):
2      '''
3       y = Qfunct(x) evalúa la función Q en x.
4      Donde y = 1/sqrt(2*pi) * integral desde x hasta inf de exp(-t^2/2) dt
5      '''
6      from scipy.special import erfc
7      from numpy import sqrt
8      y=(1/2)*erfc(x/sqrt(2))
9      return y
```

En la Figura 1.21 se muestra una representación de la función $\mathrm{Q}\left(\sqrt{x}\right)$ para valores de x comprendidos entre 0 y 15 dB.

Las variables definidas dentro de una función son *variables locales*, en el sentido de que son inaccesibles fuera de la función y que no interfieren con variables del mismo nombre definidas en otras funciones o partes del programa. Se puede decir que pertenecen al propio espacio del trabajo de la función. En contraposición, las variables definidas en un *script* son *globales*.

Al definir una función en Python, se pueden iniciar parámetros por defecto, facilitando la llamada a la función, ya que no habría que pasarle estos parámetros a no ser que sea necesario.

Al ejecutar `help(Qfunct)` se obtiene la ayuda de la función, que serán las primeras líneas comentadas desde la definición de la función hasta la primera línea de código. En este caso se obtendría:

```
Help on function Qfunct in module labcomdig:

Qfunct(x)
    y = Qfunct(x) evalúa la función Q en x.
   Donde y = 1/sqrt(2*pi) * integral desde x hasta inf de exp(-t^2/2) dt
```

En este ejemplo, la función `Qfunct` definida por el usuario está disponible para su importación. Esta y otras funciones, necesarias en capítulos siguientes, se irán guardando en un fichero que se denominará `labcomdig.py`. Por lo que para su importación será necesario que el fichero `labcomdig.py` se encuentre en el directorio de trabajo o en el camino de búsqueda.

1.11 Evitando errores comunes en la escritura de programas

Cuando se empieza a realizar programas en Python, reducir la carga computacional que lleva consigo su ejecución no es una prioridad. Sin embargo, a medida que se realizan programaciones más complejas, resulta conveniente depurar el código de los programas para obtener una mayor eficiencia. Para lograrla existen numerosas técnicas y en este apartado se van a repasar algunas de ellas.

1.11.1 Vectores lógicos

Aunque en las primeras etapas de aprendizaje de Python no se haga uso de ellos, los vectores lógicos juegan un papel clave en la depuración del código, aunque a veces lo hace también mucho menos comprensible.

Supóngase que se desea representar una función discontinua $f(x)$ definida en la forma:

$$f(x) = \begin{cases} \text{sen}(x), & \text{para sen}(x) > 0 \\ 0, & \text{en caso contrario,} \end{cases}$$

y representarla en el rango de valores $0 \leq x \leq 3\pi$. El siguiente código genera el gráfico mostrado en la Figura 1.13.

```
1  import numpy as np
2  from numpy import pi, sin # En este caso se han importado pi y sin para evitar
       tener que escribir np.pi y np.sin
3  import matplotlib.pyplot as plt
4  from matplotlib.pyplot import plot, axis  #Igualmente se han importado plot y
       axis para evitar tener que escribir plt.plot y plt.axis
5
6  x = np.arange(0,3*pi,pi/100)
7  y = sin(x)
8  y = y*(y>0)
9  plot(x,y)
10 axis([0, 3*pi, 0, 1.1])
11 plt.show()
```

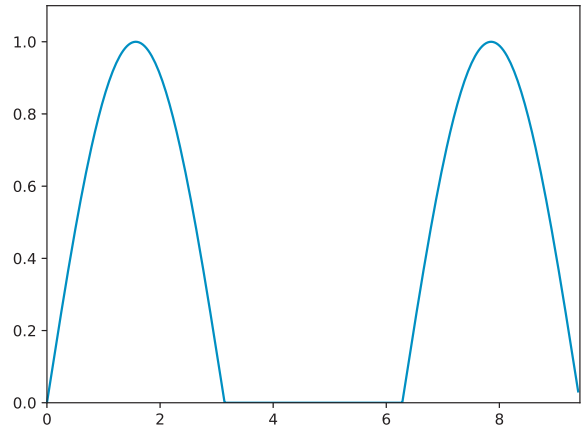

Figura 1.13: Ejemplo de Python.

Es importante detenerse en la instrucción 8 del código anterior, en la que se realiza la multiplicación de dos arrays elemento a elemento. Pero ¿qué es en realidad el array `(y>0)`? Pues simplemente un array cuyos elementos valen `True` si el correspondiente elemento del array y es mayor que cero o `False` en caso contrario. Por lo tanto, al realizar este producto se igualan a cero todos los elementos de y que son menores o iguales a cero.

Este tipo de sentencias evitan errores muy comunes, como pueden ser la división por cero, como se ve en el siguiente ejemplo. Se quiere representar la función $\mathrm{Sa}(x) = \mathrm{sen}(x)/x$ en un rango de valores que englobe el cero. En ese caso, existe una indeterminación en el origen, aunque en realidad la función vale 1 para $x = 0$, como es bien sabido. Luego si se define el array `x` en la forma: `x = −4 * pi : pi/100 : 4 * pi`, una manera de evitar que `x` tome el valor `0` consiste en sustituir este valor por el menor valor positivo de Python, definido por la constante `epsilon`, que se puede importar desde `sys.float_info`; constante que vale aproximadamente `2.2204 e − 16`. Esta asignación, junto con la representación de la función se muestra en el siguiente código y figura asociada.

```
import numpy as np
from numpy import pi, sin
import matplotlib.pyplot as plt
from matplotlib.pyplot import plot, axis
import sys

eps = sys.float_info.epsilon
x = np.arange(-4*pi,4*pi,pi/100)
x = x+(x==0)*eps
```

```
plot(x,sin(x)/x)
axis([-15, 15, -0.3, 1.1])
plt.show()
```

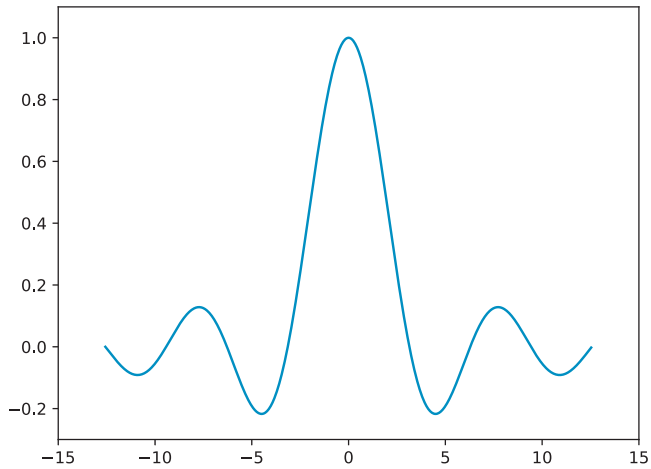

Figura 1.14: Ejemplo de Python.

Un ejemplo adicional que muestra como evitar puntos conflictivos en las definiciones puede encontrarse en el siguiente código, en el que se representan dos versiones de la función $y = \tan(x)$: en la Figura 1.15-(a) sin limitar el eje y y en la Figura 1.15-(b) se realiza una limitación en el eje y, lograda con la sentencia de la línea 12 del bloque de código siguiente:

```
1  import numpy as np
2  from numpy import pi, tan
3  import matplotlib.pyplot as plt
4  from matplotlib.pyplot import plot, grid, subplot
5  import sys
6
7  eps=sys.float_info.epsilon
8  x = np.arange(-(3/2)*pi,(3/2)*pi,pi/100)
9  y = tan(x);
10 subplot(1,2,1), plot(x,y)
11 grid(True)
12 y = y*(np.abs(y)<1e10);
13 subplot(1,2,2), plot(x,y)
14 grid(True)
15 plt.show()
```

Un último ejemplo permite mostrar un comando de Python que se utiliza para calcular el tiempo de ejecución de un programa: el comando `time()` de la librería `time`. En el siguiente código se generan muestras de un vector aleatorio que sigue

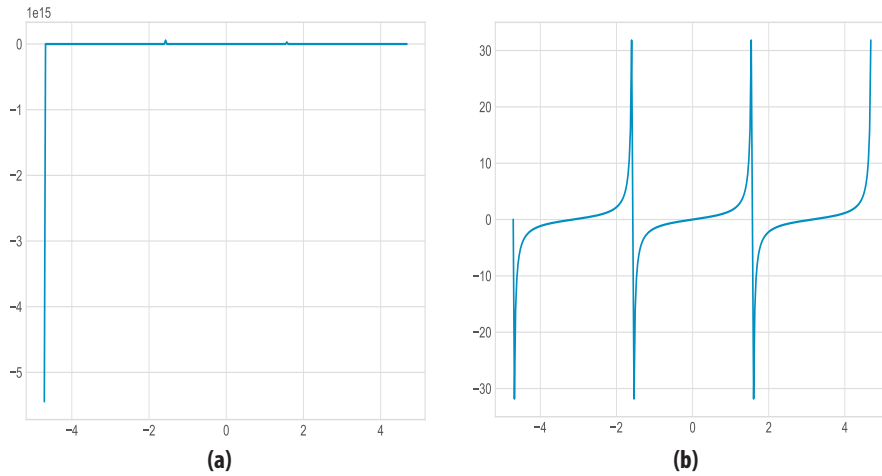

Figura 1.15: Ejemplo de Python.

una función densidad de probabilidad gaussiana de media 1 y varianza 1 y se desea contar cuántos de los elementos del vector son mayores que cero. Una primera posible programación sería la siguiente:

```
from numpy import floor
from numpy.random import randn, seed
from time import time
from labcomdig import Qfunct #Se importa Qfunct, implementada en apartados
     anteriores y salvada en fichero labcomdig.py

seed(42)
contador=0
N=10**7
r = randn(N)+1

startTime = time()
for k in r:
    if k>0:
        contador=contador+1
t = time()-startTime
teorico = floor((1-Qfunct(1))*N)
print('mayores que 0: ' + str(contador))
print('mayores que 0 (teóricos): ' + str(teorico))
print('tiempo transcurrido: ' + str(t))
```

cuya ejecución daría:

```
mayores que 0:  8415184
mayores que 0 (teóricos): 8413447.0
```

```
tiempo transcurrido: 3.7484278678894043
```

Una alternativa a esta programación se muestra a continuación:

```
from numpy import floor, sum
from numpy.random import randn, seed
from time import time
from labcomdig import Qfunct # Importa Qfunct implementado en apartados anteriores
     . Debe estar salvada en fichero labcomdig.py

seed(42)
contador = 0
N = 10**7
r = randn(N)+1

startTime = time()
contador=sum(r>0) #Importante usar sum de numpy
t = time()-startTime
teorico = floor((1-Qfunct(1))*N)
print('mayores que 0: ' + str(contador))
print('mayores que 0 (teóricos): ' + str(teorico))
print('tiempo transcurrido: ' + str(t))
```

obteniéndose el siguiente resultado:

```
mayores que 0:  8415184
mayores que 0 (teóricos): 8413447.0
tiempo transcurrido:  0.43319201469421387
```

que representa aproximadamente un 11% del tiempo de ejecución anterior. Es decir, este segundo código es más eficiente. La razón hay que buscarla en que en este segundo código se ha evitado la utilización del bucle `for`. En general, se debe evitar siempre que se pueda la utilización de bucles en la programación, sustituyéndolos por expresiones con vectores (arrays) lógicos.

1.11.2 Otras técnicas de optimización

Puesto que los scripts de Python son ejecutados línea a línea, Python consume tiempo en comprobar la sintaxis de los mismos.

Aunque Python oculta el procedimiento de asignar memoria para las variables que se definan, este hecho consume una gran cantidad de tiempo. Por ello, es aconsejable preasignar espacio a las variables que se definan mediante su definición con vectores adecuados.

En el siguiente ejemplo se implementan dos alternativas para sumar los números primos menor que un número dado. Se deja como tarea al lector que tratar de entender cómo funcionan ambas posibilidades.

```
# Alternativa con bucle for
import numpy as np
```

```
from time import time
from sympy import primerange #Es posible que tenga que instalar la librería SymPy

tic=time()
N=1000;
s = 0;
for j in primerange(1,N):
    s = s + j

print('suma de los ' +str(N) +' primeros números primos: ' +str(s))
print('tiempo transcurrido   :' +str(time()-tic))

# Alternativa sin bucle for
startTime = time()
s = np.sum(list(primerange(1,N)));

print('suma de los ' +str(N) +' primeros números primos: ' +str(s))
print('tiempo transcurrido   :' +str(time()-startTime))
```

Si se ejecuta el código mostrado, puede observarse que la alternativa sin bucle `for` es más rápida. Aún es más evidente cuando se ejecuta el código que realiza la multiplicación de dos matrices aleatorias:

```
from numpy.random import randint
from numpy import zeros
from time import time

# Alternativa con bucle for
N = 100
M = 100
x = randint(10,size=(M,N))
y = randint(10,size=(N,M))
z = zeros([M,M])
tic = time()
for k in range(M):
    for l in range(M):
        z[k,l]=np.sum(x[k,:]*y[:,l])

print('tiempo transcurrido  : ' +str(time()-tic))

# Alternativa sin bucle for

tic = time()
z = np.dot(x,y)
print('tiempo transcurrido  : ' +str(time()-tic))
```

con el resultado que se muestra a continuación:

```
tiempo transcurrido  :   0.10498189926147461
tiempo transcurrido  :   0.0006690025329589844
```

Es decir, varios órdenes de magnitud más rápida la opción sin bucles que con bucles.

Para finalizar este breve resumen de técnicas de optimización de Python, se muestran dos funciones que implementan la célebre criba de Eratóstenes para la búsqueda de números primos menor que uno dado. Es interesante que el alumno comprenda cada uno de los códigos que siguen para poder utilizar las técnicas que se presentan.

Código 1.3: Criba de Eratóstenes. Versión 1.

```
1  def criba01(lastNumber):
2      from numpy import mod, where, array
3
4      List = list(range(2, lastNumber))
5      primeList = [];
6
7      while(List[0]**2 <lastNumber):
8
9          primeList.append(List[0])
10         List = array(List)
11
12         List = list(List[where(mod(List,List[0]))])
13
14     primeList.extend(List)
15     return primeList
```

Código 1.4: Criba de Eratóstenes. Versión 2.

```
1  def criba02(x):
2      from numpy import array, sqrt, arange
3      P = list(range(x))
4      P = array(P)
5      for n in range(2,int(sqrt(x))):
6         if P[n]:
7            P[arange(2*n,x,n)] = 0
8      P = P[P != 0]
9      return list(P)
```

1.12 Simulación discreta de una señal de energía de duración finita

En este texto el objetivo es aplicar Python en el ámbito de la ingeniería, y en especial a comunicaciones digitales. A modo de ejemplo, en lo que resta de este capítulo, se aplica lo explicado anteriormente a un problema concreto, de gran relevancia, en la intersección entre procesado de señal y comunicaciones digitales. La pregunta que se formula es la siguiente: ¿cómo representar de manera experimental (es decir, de forma discreta) una señal de duración finita y energía dada? Quizás el problema se entienda mejor con un ejemplo.

Se desea representar la señal de duración T, $g(t)$, definida en la forma:

$$g(t) = \begin{cases} 1 & 0 \leq t \leq T/2 \\ -1 & T/2 \leq t \leq T \end{cases} \tag{1.1}$$

y mostrada en la Figura 1.16. Si se supone que $T = 4$ ms (no se hace igual a 1 segundo

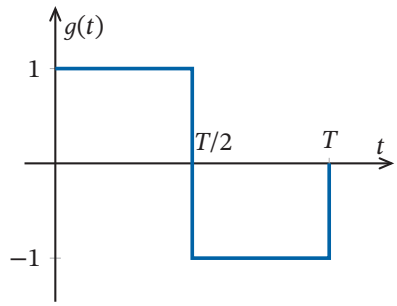

Figura 1.16: Pulso de duración finita.

para evitar confusiones), la energía de este pulso vendrá dada por:

$$E_g = \int_{-\infty}^{\infty} g^2(t)\,\mathrm{d}t = \int_0^T \mathrm{d}t = 4 \cdot 10^{-3}\ \mathrm{J}.$$

Para representar esta señal en Python, se ejecuta el Código 1.5 que generaría la señal mostrada en la Figura 1.17.

Código 1.5: Generación del pulso de duración finita de la Figura 1.16: representación continua.

```
import numpy as np
import matplotlib.pyplot as plt

# Definición de los parámetros de la representación:
T = 4e-3  # Duración
L = 64    # Número de muestras que se van a utilizar
Tm = T/L   #Tiempo de muestreo (Intervalo de separación entre muestras)
t = np.arange(0,T,Tm)   #Eje de tiempos.

# Definición de la señal
gt = np.ones(L)   #La función evaluada en los puntos elegidos
gt[L//2:] = -1

# La representación continua
plt.figure(1)
h = plt.plot(t,gt) #Se representa gt frente t
plt.axis([0, T, -1.1, 1.1])
```

```
plt.setp(h,'linewidth',1.0)   #Permite fijar el grosor de línea
plt.ylabel('g(t)')  #Etiqueta para el eje de ordenadas
plt.xlabel('t(ms)') #Etiqueta para el eje de abscisas
plt.show()
```

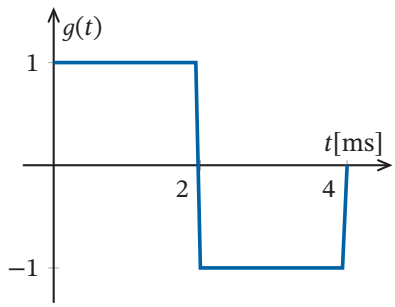

Figura 1.17: Pulso de duración finita generado con el Código 1.5.

Comparando las gráficas representadas en las figuras 1.16 y 1.17 del pulso definido en (1.1), se observa que su apariencia es similar pero cabría preguntarse por sus energías. De acuerdo con el Código 1.5, en la representación de la Figura 1.17 en realidad lo que se ha hecho es mostrar una secuencia de puntos, que de acuerdo con la instrucción `plot` se han unido entre sí. Si se quisiera representar esta secuencia, como lo que es, una secuencia de puntos, se podría ejecutar el Código 1.6 que se muestra a continuación y que daría como resultado la Figura 1.18.

Código 1.6: Generación del pulso de duración finita de la Figura 1.18: representación discreta.

```
from numpy import ones
from matplotlib.pyplot import axis, setp, ylabel, xlabel, show, stem

# Puede observarse que en este caso se han importado las funciones que van a
     utilizarse, evitando con ello tener que escribir plt.axis, plt.ylabel...

L = 64 # Número de muestras que vamos a utilizar

# Definición de la señal de forma discreta
g1n = ones(L)
g1n[L//2:] = -1

h = stem(g1n)
axis([1,L,-1.1,1.1])
setp(h,'linewidth',1.0)
ylabel('$g_1(n)$')      # Texto entre $$ formato formulas latex
xlabel('n')
show()
```

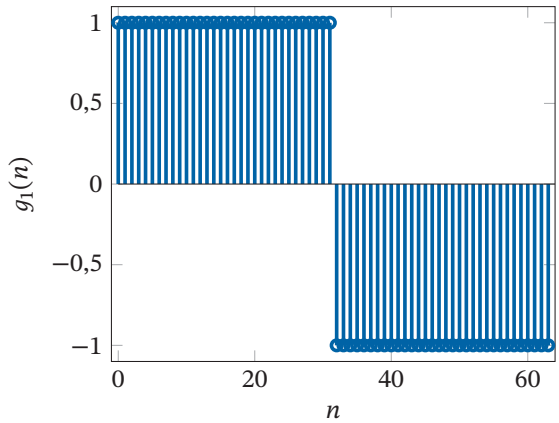

Figura 1.18: Secuencia generada con el Código 1.6.

Es decir, lo que se ha simulado y representado es la secuencia $g_1(n)$ dada por:

$$g_1(n) \stackrel{\text{def}}{=} g(t)|_{t=nT_m}, \tag{1.2}$$

estando el intervalo de separación entre las muestras (tiempo de muestreo) definido en la forma:

$$T_m = \frac{T}{L}. \tag{1.3}$$

$g_1(n)$ es una *representación fidedigna* de $g(t)$. Sin embargo, si se calcula su energía se encuentra que es igual a:

$$E_{g_1} = \sum_{n=0}^{L-1} g_1^2(n) = 64 \text{ J}. \tag{1.4}$$

Esto es, la secuencia $g_1(n)$ *no tiene la misma energía* que la señal que está simulando. ¿Cómo se puede hacer que la señal que se genere (discreta) tenga la misma energía que la señal que se está simulando (analógica)? La respuesta se encuentra fácilmente sin más que calcular la energía de $g(t)$ de manera aproximada. E_g vendría dada por:

$$E_g = \int_0^T g^2(t)\,\mathrm{d}t \approx T_m \sum_{n=0}^{L-1} g^2(nT_m). \tag{1.5}$$

Se puede ver que se cumple:

$$E_g = T_m E_{g_1}.$$

Por lo tanto, si se desea que la señal discreta que se utiliza para representar y simular una señal continua tenga la misma energía que la señal que representa, es suficiente

con definir una secuencia $g_2(n)$ en la forma:

$$g_2(n) \stackrel{\text{def}}{=} \sqrt{T_m}\, g(t)|_{t=nT_m}, \tag{1.6}$$

de forma que su energía viene dada por:

$$E_{g_2} = \sum_{n=0}^{L-1} g_2^2(n) = T_m \sum_{n=-0}^{L-1} g^2(nT_m) = E_g. \tag{1.7}$$

En este laboratorio, en general, cuando se proceda a simular una señal continua mediante una señal discreta se definirá una secuencia discreta de acuerdo con la ecuación (1.6). En el siguiente código se genera esta señal discreta que tiene exactamente la misma energía que la señal continua que representa y cuyo resultado se muestra en la Figura 1.19.

Código 1.7: Generación del pulso de duración finita de la Figura 1.19: representación discreta *normalizada*.

```
from numpy import ones, sqrt, arange
from matplotlib.pyplot import stem, axis, setp, ylabel, xlabel

# Se definen los parámetros de la representación:
T = 4e-3 # Duración de la representación
L = 64       # Número de muestras que vamos a utilizar en la representación
Tm = T/L      # Tiempo de muestreo (Intervalo de separación entre muestras)
t = arange(0,T,Tm) # Eje de tiempos.

# Definicion de la señal continua
gt = ones(L) # La función evaluada en los puntos elegidos
gt[L//2:] = -1

# La secuencia discreta normalizada
g2n = sqrt(Tm)*gt

# Su representación
h = stem(g2n)
axis( [-1, L+1, -sqrt(Tm)*1.1, sqrt(Tm)*1.1])
setp(h,'linewidth', 1.0)
ylabel('g_{2}(n)')
xlabel('n')
```

En ocasiones resulta mucho más simple realizar un diseño directamente en tiempo discreto y con posterioridad obtener el equivalente continuo. Es decir, se realiza el diseño discreto de acuerdo con las especificaciones pedidas y posteriormente se le da la vuelta a la ecuación (1.6) de manera que si se llama $g(n)$ a la secuencia diseñada,

$$g(nT_m) = \frac{1}{\sqrt{T_m}}\, g(n). \tag{1.8}$$

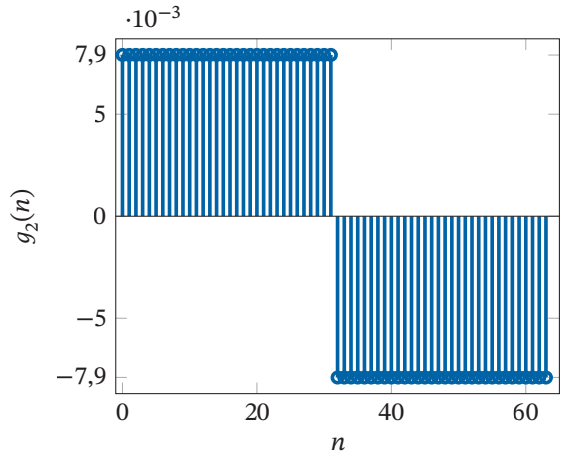

Figura 1.19: Secuencia normalizada generada con el Código 1.7.

Supóngase, por ejemplo, que se desea simular un pulso rectangular de duración $T = 4$ ms y energía $E = 4$ J. En este caso, si se parte de una secuencia discreta, con L puntos, $g(n)$, definida en la forma:

$$\texttt{gn} = \texttt{np.ones(L)}, \tag{1.9}$$

es fácil comprobar que su energía vendría dada por $E_g = L$ J, por lo tanto, una señal discreta como la pedida, con energía $E = 4$ J sería:

$$\texttt{gn} = \sqrt{\frac{\texttt{E}}{\texttt{L}}}\texttt{ones(L)} \tag{1.10}$$

donde se observa que se divide por $\sqrt{L}$, normalizando a energía unidad, y se multiplica por $\sqrt{E}$, para que tenga la energía buscada. La señal analógica deseada sería, en este caso concreto:

$$\texttt{gt} = \frac{1}{\sqrt{\texttt{Tm}}} * \texttt{gn} = \frac{1}{\sqrt{\texttt{T/L}}} * \texttt{gn} = \sqrt{\frac{\texttt{E}}{\texttt{T}}} * \texttt{ones(L)}. \tag{1.11}$$

En el Código 1.8 se muestra por último todo lo anterior y su ejecución conduce a la representación de la Figura 1.20.

Código 1.8: Señal discreta y señal continua.

```
import numpy as np
import matplotlib.pyplot as plt

# Parámetros de diseño
E = 4
```

```
T = 4e-3
L = 8

# Tiempo entre muestras
Tm = T/L

# Señal discreta de energía E
sn = np.ones(L)
En = sn@sn #Energía del pulso sin normalizar
sn = np.sqrt(E/En)*sn

# Energía de la señal discreta
msg = 'Energía de la señal discreta sn: %.2f J'%(sn@sn)
print (msg)

# Definición de la señal continua
st = np.sqrt(1/Tm)*sn

# Energía de la señal continua
msg = 'Energía de la señal continua st: %.2f J'%(st@st*Tm)
print (msg)

# Representación de las dos señales
plt.figure(1)
plt.subplot(1,2,1), plt.stem(sn, use_line_collection=True)
plt.xlabel('n'), plt.ylabel('sn(n)')
plt.axis([0,L+1, 0, max(sn)*1.1])
t = np.arange(0,T,Tm)
plt.subplot(1,2,2), plt.plot(t,st)
plt.xlabel('t[s]'), plt.ylabel('st(t)')
plt.axis([0, T, 0, max(st)*1.1])
plt.tight_layout()
plt.show()
```

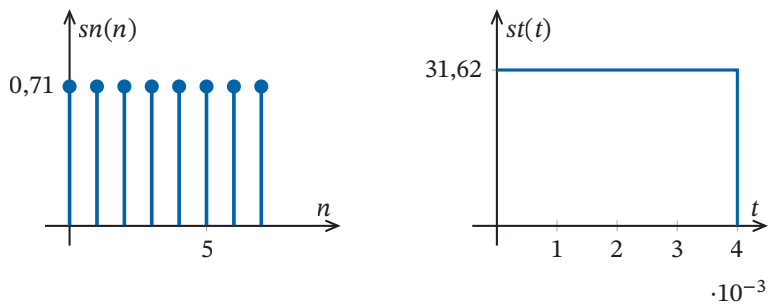

Figura 1.20: Gráfica generada con el Código 1.8.

Ejercicios propuestos

1.1 Realizar las operaciones de extracción de columnas análogas a las realizadas en el Ejemplo 1.5.1.

1.2 En las siguientes expresiones se asignan valores lógicos a la variable `x`. Calcular en primer lugar el valor que se le asigna y posteriormente comprobar la respuesta en Python.

a) $x = 3 > 2$
b) $x = 2 > 3$
c) $x = 1 < 1$
d) $x = -4 <= -3$
e) $x = 2! = 2$
f) $x = 3 == 3$
g) $x = 0 < 0.5 < 1$

1.3 Supóngase que se han definido los siguientes vectores en Python:

$$a = \begin{bmatrix} 2. & -1. & 4. & 0. \end{bmatrix}$$
$$b = \begin{bmatrix} 3. & 2. & -1. & 4. \end{bmatrix}$$

Encontrar de forma manual el valor del vector `c` definido en el conjunto de operaciones siguiente. Comprobar con Python la corrección de las respuestas.

a) `c = a - b`
b) `c = b + a - 3`
c) `c = 2 * a + a * *b`
d) `c = b/a`
e) `c = b * a`
f) `c = a * *b`
g) `c = 2 * *b + a`
h) `c = 2 * b/3 * *a`
i) `c = b * 2 * a`

1.4 Dados los vectores `x` e `y` de 10^6 elementos, escribir un código que nos devuelva el primer valor en el que `x` es mayor que `y`.

1.5 Diseñar un script que genere números aleatorios uniformemente distribuidos entre $[-2,2]$.

1.6 Obtener la representación mostrada en la Figura 1.21 haciendo uso de la función `Qfunct` (definida por el usuario) y `plt.semilogy`.

1.7 Diseñar una función cuyo argumento de entrada sea un número entero positivo y cuya salida nos dé la suma de todos los primos menores o iguales a ese número. Utilizar alguna de las versiones de la criba de Eratóstenes analizadas.

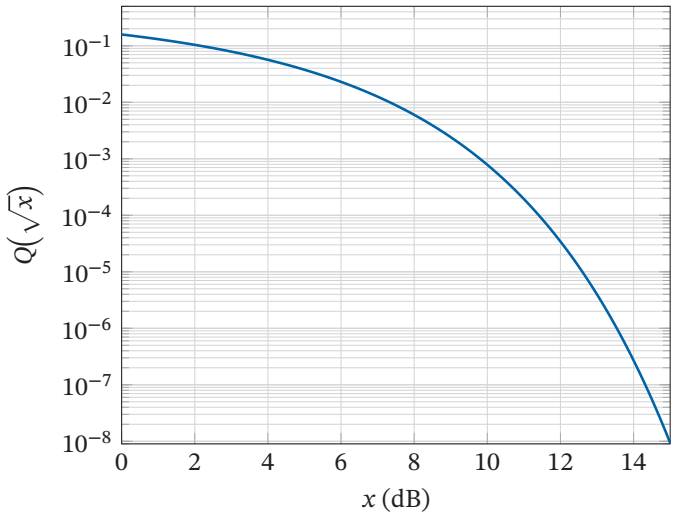

Figura 1.21: Función $\mathrm{Q}(\sqrt{x})$.

1.8 Diseñar una función cuyo argumento de entrada sea un número entero positivo y cuya salida dé la suma de todos los números impares menores a ese número. Utilizar la función `mod` de `NumPy`. Puede resultar interesante crear un procedimiento similar a la criba de Eratóstenes.

1.9 Diseñar una función con un número de argumentos variables: frecuencia, instante inicial e instante final y, opcionalmente, periodo de muestreo. En la ejecución de la función, se representará una señal sinusoidal de frecuencia la dada, amplitud unidad, entre los instantes dados y, si no existe un periodo de muestreo de la representación, un periodo de muestreo igual a 1/10 del periodo de la sinusoide. Se desea obtener adicionalmente los instantes en los que la señal se está representando.

1.10 Diseñar un script que defina y represente las funciones indicadas para valores de t comprendidos entre $[-5,5]$.

a) $y = 3t$.
b) $y = 4t^2$.
c) $y = \texttt{rect}(t/2)$.
d) $y = \texttt{tri}(t)$.

1.11 Repetir lo anterior pero garantizando que las señales discretas tengan la misma energía que las correspondientes señales continuas.

1.12 Haciendo uso de la función `np.convolve`, diseñar una función que, dados dos vectores, realice una convolución[15] de los mismos. La función debe generar

[15]Se refiere a una convolución discreta.

una representación de los dos vectores de entrada así como una representación de la convolución, todo dentro de una única figura. Los ejes deben elegirse de manera que todos los puntos queden representados. También se desea conocer la duración de la señal de salida de la convolución.

1.13 Dado un vector de entrada, diseñar una función que nos devuelva su energía. ¿Se podría hacer uso de la función `np.correlate`? Justificar la respuesta.

Ejercicio Práctica 1

Como se mencionó en el Prefacio, en cada una de las prácticas de la asignatura de Comunicaciones Digitales, se plantea un Ejercicio que debe ser resuelto por los alumnos en la sesión de práctica correspondiente. En esta primera Práctica se desean realizar las siguientes tareas:

1. Generar una secuencia A_n de $N = 10$ números aleatorios de valores ± 2.
2. Generar un pulso rectangular de energía unidad, $s(t)$, representado mediante $L = 8$ muestras.
3. Representar la función que se obtiene si el pulso $s(t)$ se multiplica por la secuencia A_n y se desplaza cada T_b s. Esto es, obtener y representar $X(t)$ definido por:

$$X(t) = \sum_{k=0}^{9} A_k s(t - kT_b).$$

   Mostrar explícitamente los valores máximos y mínimos que toma la señal así como su duración en segundos.
4. ¿Cuál ha sido la energía total transmitida? ¿Y la potencia?
5. Repetir lo anterior para un pulso senoidal de energía unidad $s(t)$, proporcional al mostrado en la Figura 1.12, de duración $T_b = 10^{-3}$s, representado mediante $L = 8$ muestras.

Los apartados **2.** y **3.** tienen que hacerse en tiempo discreto y en tiempo continuo y convertirlo en el que no se haga originalmente. Valorar la ventaja de hacerlo en un sentido u otro y, en cualquier caso, obtener las respuestas en los dos marcos temporales.

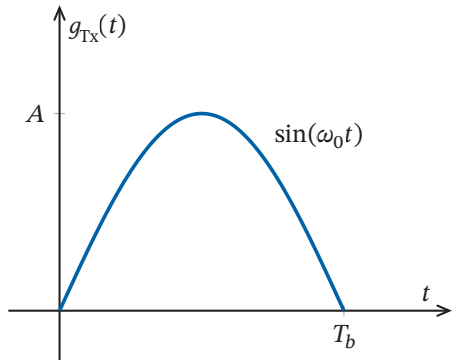

Figura 1.22: Pulso senoidal genérico.